Branch Prediction Models Exploration

Daniela Trevina
Department of EECS
Texas A&M University-Kingsville
Kingsville, USA
daniela.trevino@students.tamuk.edu

Daniela Lopez
Department of EECS
Texas A&M University-Kingsville
Kingsville, USA
daniela.lopez@students.tamuk.edu

Mengxiang Jiang
Department of EECS
Texas A&M University-Kingsville
Kingsville, USA
mengxiang.jiang@students.tamuk.edu

Samah Allahyani
Department of EECS
Texas A&M University-Kingsville
Kingsville, USA
samah.allahyani@students.tamuk.edu

Ugochukwu Onyeakazi
Department of EECS
Texas A&M University-Kingsville
Kingsville, USA
ugochukwu.onyeakazi@students.tamuk.edu

***Abstract*—Due to the importance of branch prediction, running benchmark simulations is a good way to understand and evaluate different models. This paper discusses six different models (bimodal, gshare, perceptron, hashed perceptron, TAGE, LTAGE), and tests them using ChampSim on different SPEC benchmarks. The results are analyzed with further work suggested.**

I. Introduction

Branch prediction is a crucial component in modern computer architecture to improve performance and energy efficiency. When executing a program, the processor encounters branches in the control flow, where it must decide which path to take based on a conditional statement. Incorrect branch predictions result in wasted computation cycles and negatively impact the overall performance of the system. To mitigate this issue, branch predictors are employed to predict the outcome of the conditional statements, and enable the processor to pre-fetch instructions from the predicted path, reducing the overhead of branching.

Here is a concrete example showcasing how even small improvements in branch prediction accuracy can result in huge performance improvments, from Onur Mutlu [6]. Assume the processor has 20 pipeline stages, with a 5-wide fetch. If 1 out of 5 instructions that are fetched is a branch (distributed uniformly), then here are time lengths to fetch 500 instructions and resulting instructions per cycle for different prediction accuracies:

1) 60%:
   100 + 20 * 40(branch penalty) = 900 cycles
   IPC = 500/900 = 0.555

2) 90%:
   100 + 20 * 10(branch penalty) = 300 cycles
   IPC = 500/300 = 1.666

3) 99%:
   100 + 20 * 1(branch penalty) = 120 cycles
   IPC = 500/120 = 4.166

4) 100%:
   100 cycles
   IPC = 500/100 = 5

As one can see, going from 60% accuracy to 90% results in over 300% performance improvement, and from 90% to 99% is similar.

Over the years, a variety of branch prediction models have been developed to improve the accuracy of predictions. These models range from static branch predictors that use heuristics to predict the outcome of branches, to more sophisticated dynamic branch predictors (note: all models tested in this paper are dynamic) that rely on past history to make predictions. Furthermore, the emergence of machine learning and artificial intelligence has led to the development of more advanced branch prediction techniques for both static and dynamic branch predictors.

The aim of this paper is to explore the some branch prediction models and compare their performances. We will examine the underlying principles of each model, the algorithms used to make predictions and their implementations in a simulated processor.

II. Models

In this section, we will cover the various branch predictor models used in the simulation.

A. Bimodal Predictor

This model was first proposed by James E. Smith as an improvement on the single-bit last-time predictor [12]. This is a local history predictor in the sense that the Program Counter (PC) is used to differentiate branches and their past outcomes. It stores two bits in the branch target buffer (BTB) indicating whether this particular branch was taken or not taken in the last two times encountered. These two bits encode a state machine with four states shown in Fig. 1. The *Strongly Taken* and

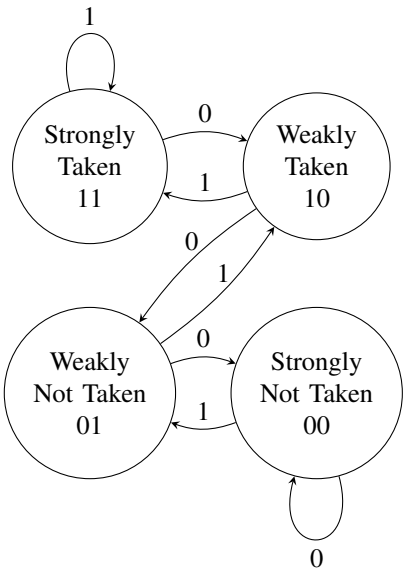

Fig. 1. Bimodal Predictor

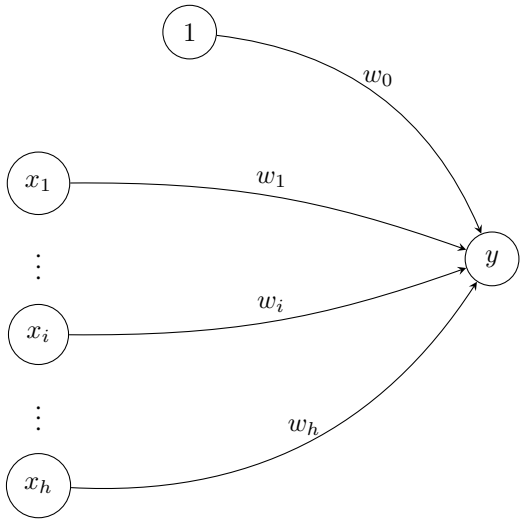

Fig. 2. Perceptron Predictor

Weakly Taken states will predict *taken* for the current branch, while the *Strongly Not Taken* and *Weakly Not Taken* states will predict *not taken*.

B. Gshare Predictor

This model was first proposed by Scott McFarling as an improvement on the global history predictor [5]. A normal global history predictor uses the single history from the Global History Register (GHR) to index into the Pattern History Table (PHT) in order to predict all branch outcomes regardless of the PC. Gshare XORs the PC with the GHR for indexing into the PHT, which adds some local context information for the predictor to take into account.

C. Perceptron Predictor

This model is a single-layer version of an artificial neural network that can identify and classify patterns, first applied to branch prediction by Daniel A. Jimenéz and Calvin Lin [3]. In the model shown in Fig. 2 are the input vector (x), the weight vector (w), and the output (y). The inputs correspond to values taken from the global history register, with the exception of x_0, the bias, always set to 1. The output's sign determines the prediction. If the sign is negative, the branch is not taken, otherwise it is taken. Equation (1) shows the calculation of the output:

$$y = w_o + \sum_{i=1}^{h} x_i w_i \tag{1}$$

The weights of the perceptron are trained using the algorithm below:

```
if sgn(y) != t or abs(y) <= theta  then
    for i := 0 to n do
        w[i] := w[i] + t*x[i]
    end
end
```

t is 1 if the branch is actually taken or -1 if not. θ is the threshold to determine when training should stop. w_i is incremented if t and x_i agree, and decremented if they disagree.

D. Hashed Perceptron (HP) Predictor

This predictor is based off of the perceptron branch predictor from the previous section(II-C) but with a few important changes. The first change is instead of a single table for the weights, multiple independently indexed tables of perceptron weights are used, introduced by Jimenéz [2]. The second is using the hash of the global history rather than just the global history for indexing in order to reduce the number of independent tables, contemporaneously by Seznec [8], Tarjan and Skadron [11], and Loh and Jimenéz [4]. The third is instead of using a fixed global history length, exponentially increasing global history lengths are used for indexing, by Seznec [7] [9]. This idea is actually one of the key parts of the TAGE predictor described in the next section(II-E). The last is to dynamically adjust the θ value for the training, also by Seznec [7] [9].

E. TAGE Predictor

TAGE stands for TAgged GEometric history length branch predictor, and it was introduced by André Seznec [10]. The first key idea, as mentioned earlier, is using exponentially increasing global history lengths for indexing into multiple tables, in this case PHTs while for the hashed perceptron it was the weights. The second is to intelligently allocate the PHT entries to different branches. Fig. 3 shows a 3-component TAGE predictor, while the TAGE predictor evaluated in our simulations has 13 components.

F. LTAGE Predictor

This predictor is based off of the TAGE predictor from the previous section(II-E), but with the addition of a loop predictor. It was introduced by Seznec and won the 2nd

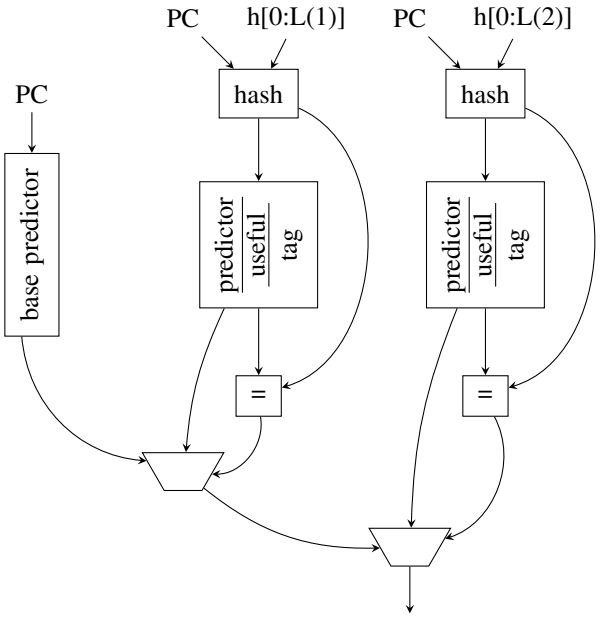

Fig. 3. 3-Component TAGE Predictor

Championship Branch Prediction competition [8]. A loop predictor first predicts the limit on how many iterations the loop will execute. It keeps track of the number of times the loop as executed, and as long as this count is below the limit, it will predict taken.

III. Methodology

The study conducts experiments using the ChampSim Simulator. Each predictor is implemented in the branch prediction unit of the simulator. The predictors are then tested on application benchmarks from Standard Performance Evaluation Corporation (SPEC) suite of benchmarks. The performance of each predictor is recorded and presented in the next section (IV).

A. Simulation Environment

ChampSim (Championship Simulator) uses a modular design and configurable structure to achieve a low barrier to entry into the field of microarchitectural simulation [1].

ChampSim operates by analyzing program behavior through a process called tracing, which involves instrumenting and running the program offline to produce a summary of its activity. This trace can be saved and later used as a substitute for actual program execution during simulation. Although this method sacrifices some accuracy, particularly in terms of operating system interactions, it offers advantages in terms of reproducibility and simulation speed.

ChampSim consists of four types of modules: branch predictors, branch target buffers (BTBs), memory prefetchers, and cache replacement policies. Each cache and Translation Lookaside Buffer (TLB) also has its own prefetcher and replacement policy. These modules may be different across different cache and TLB levels, and even between cores. During configuration, ChampSim identifies all necessary modules and links their source as required. For our simulations, we only change which branch predictor to use in the configuration, leaving everything else in their default settings.

B. Benchmarks

The Standard Performance Evaluation Corporation (SPEC) provides benchmark suites to evaluate performance and energy efficiency for computing systems. These benchmarks are designed to measure various aspects of computer performance, such as CPU speed, memory bandwidth, graphics performance, etc. This study makes use of four application benchmarks from the SPEC CPU benchmark suite. *gcc* and *perl* are used for testing integer arithmetic (SPECint). *soplex* and *povray* are used for testing floating point performance (SPECfp).

C. Metrics

The performance of the branch predictors are evaluated using the following metrics:

1) *Instructions Per Cycle* (IPC): IPC is a measure of the average number of instructions executed in each clock cycle of the CPU. It is calculated by dividing the total number of instructions executed during a given time by the number of clock cycles used to execute those instructions. A higher IPC value indicates that the processor is able to execute more instructions per cycle, which translates into higher performance.

$$IPC = \frac{Number\ of\ Instructions\ Committed}{Simulation\ Time\ in\ Cycles}$$

2) *Mispredictions Per Kilo Instructions* (MPKI): MPKI is a performance metric used to measure the accuracy of a processor's branch predictor. It represents the number of mispredicted branches per thousand instructions executed. It is calculated by dividing the total number of mispredicted branches by the total number of instructions executed, and then multiplying by 1000 to express the result in terms of mispredictions per thousand instructions. A lower MPKI value indicates better branch prediction accuracy and can lead to higher processor performance.

$$MPKI = \frac{Number\ of\ Mispredictions}{Number\ of\ Instructions\ Committed} \times 100$$

3) *Accuracy*: Accuracy is the percentage of correct predictions made by the branch predictor. It is computed by dividing the number of correctly predicted branches by the total number of predicted branches and multiplying the result by 100. A higher accuracy value indicates that the branch predictor's predictions are closer to the actual branch outcomes, and thus the model's performance is more reliable.

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Number\ of\ Branches\ Committed} \times 100$$

IV. Results and Discussion

Each model was executed in the ChampSim simulator and the results for four benchmarks from the SPEC CPU benchmark suite were recorded. Table I shows the performance of each model across the various benchmarks. In this section, we present a detailed discussion of the results obtained.

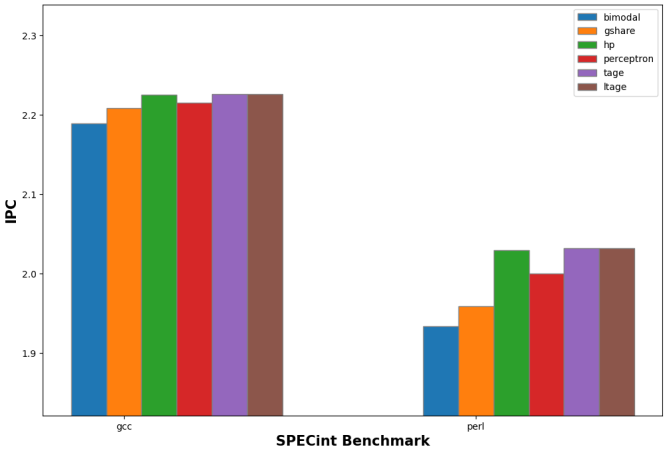

Fig. 4. IPC Results on Integer Benchmarks (all bar graphs made in *python* using *matplotlib*)

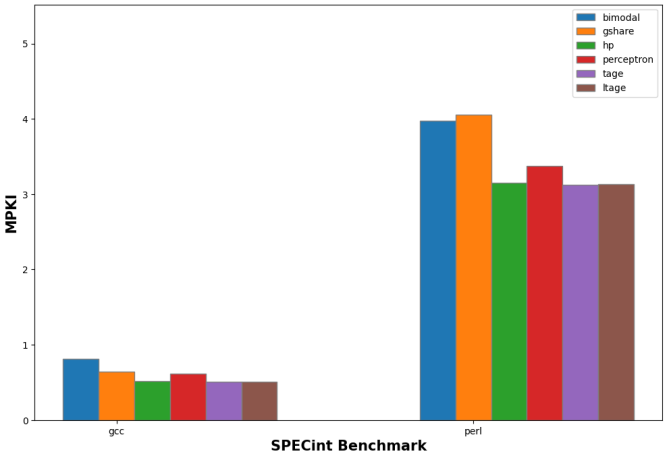

Fig. 5. MPKI Results on Integer Benchmarks

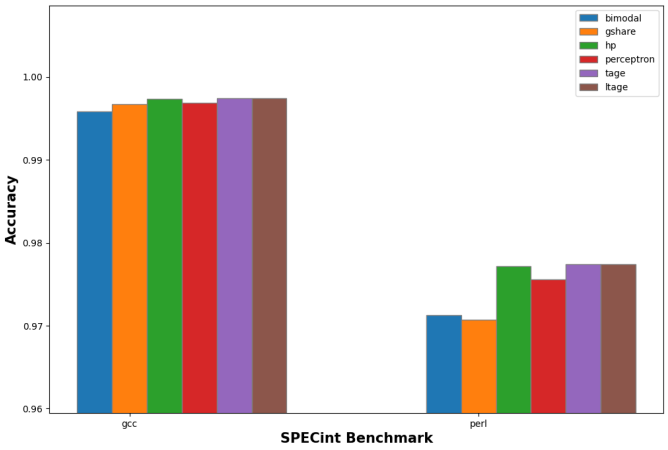

Fig. 6. Accuracy Results on Integer Benchmarks

Fig. 4, Fig. 5 and Fig. 6 show that for integer arithmetic testing, the HP Predictor, TAGE Predictor, and LTAGE Predictor showed the best performance in terms of IPC and accuracy for both gcc and perl benchmarks. Specifically, the TAGE Predictor displayed the highest IPC and accuracy for the gcc and perl benchmarks. It is worth noting that the Bimodal Predictor had the lowest IPC in gcc and perl. In terms of accuracy, the Gshare predictor had the lowest in the perl benchmark while the Bimodal had the lowest in gcc.

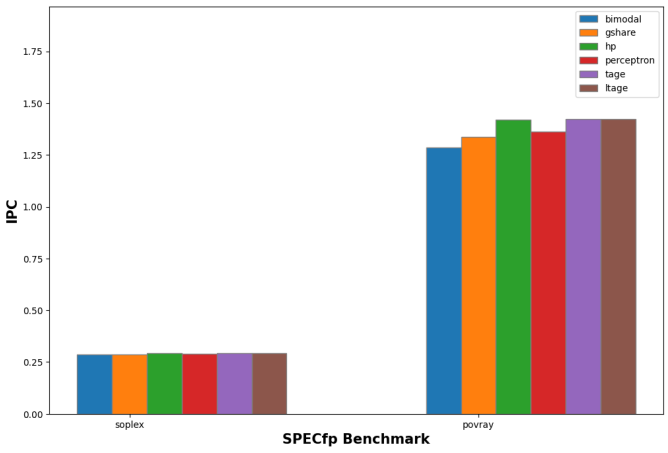

Fig. 7. IPC Results on Floating Point Benchmarks

Fig. 7, Fig. 8 and Fig. 9 show that the results for the floating point performance testing, the HP Predictor demonstrated the best performance in terms of IPC and accuracy in the soplex benchmark. Whereas in povray benchmark, the LTAGE demonstrated the best performance in both IPC and accuracy. TAGE Predictor and LTAGE Predictor showed very similar performance to each other, with only slight differences in their IPC and accuracy values. On the other hand, the Bimodal Predictor and Gshare Predictor exhibited the lowest performance in terms of IPC and accuracy for the soplex and povray benchmarks, respectively.

TABLE I
PERFORMANCE RESULTS

	Benchmark: gcc			Benchmark: perl			Benchmark: soplex			Benchmark: povray		
Predictors	**IPC**	**MPKI**	**Accuracy**	**IPC**	**MPKI**	**Accuracy**	**IPC**	**MPKI**	**Accuracy**	**IPC**	**MPKI**	**Accuracy**
bimodal	2.18893	0.8116	0.995865	1.9339	3.97542	0.971272	0.287601	12.9765	0.928403	1.28419	10.5247	0.927332
gshare	2.20866	0.6419	0.996729	1.95874	4.05482	0.970698	0.287169	12.9904	0.928326	1.33658	8.4325	0.941778
hp	2.22541	0.51766	0.997362	2.02928	3.15328	0.977213	0.292012	10.9323	0.939681	1.41964	6.73906	0.95347
perceptron	2.21491	0.61248	0.996879	2.00043	3.3734	0.975622	0.288693	12.6598	0.93015	1.36391	7.75706	0.946441
TAGE	2.22633	0.5082	0.997411	2.03183	3.12708	0.977402	0.291466	11.2682	0.937828	1.42276	6.65282	0.954066
LTAGE	2.22616	0.50936	0.997405	2.03162	3.12894	0.977389	0.291411	11.2704	0.937816	1.4228	6.65236	0.954069

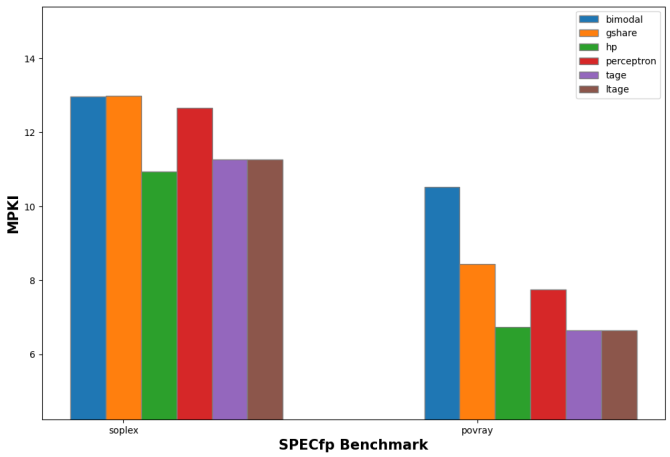

Fig. 8. MPKI Results on Floating Point Benchmarks

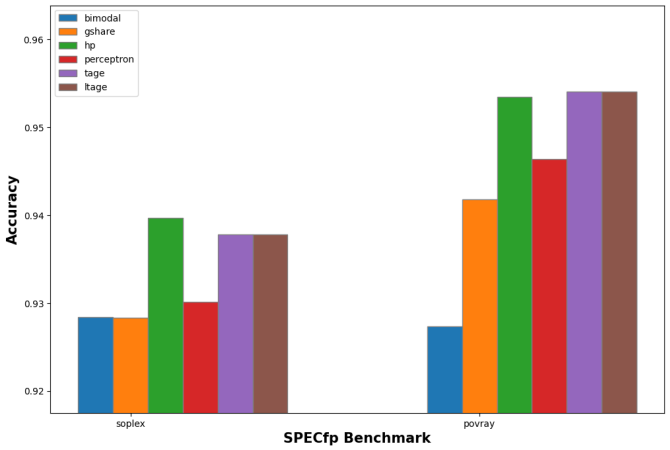

Fig. 9. Accuracy Results on Floating Point Benchmarks

In terms of MPKI, the TAGE Predictor had the lowest MPKI values for two out four benchmarks, indicating fewer misses per thousand instructions compared to the other predictors. The Bimodal Predictor had the highest MPKI values for gcc and povray benchmarks, while the Gshare Predictor had the highest MPKI for the perl and soplex benchmark.

Overall, the HP Predictor, TAGE Predictor, and LTAGE Predictor consistently outperformed the other predictors in terms of IPC, accuracy, and MPKI across all four benchmarks. Among these three predictors, the TAGE Predictor showed the best performance for integer arithmetic performance testing while the LTAGE and HP showed the best performance for floating point performance testing. Note that the Bimodal Predictor and Gshare Predictor demonstrated the weakest performance in most cases, though they are much simpler in design and therefore cost less hardware.

V. CONCLUSION

This paper explored some branch prediction techniques through implementation as well as a comparative study using simulation experiments on a range of benchmark suites. The experimental results demonstrate that the HP Predictor, TAGE Predictor, and LTAGE Predictor are the most effective branch prediction models, consistently outperforming the other predictors in terms of IPC, accuracy, and MPKI across different benchmarks. The commonality of these three predicators is the use of geometric global history lengths, suggesting the importance of this idea for branch predictions.

In the future we might incorporate more advanced branch prediction schemes such as TAGE-SC-L in our framework for comparison. Also, we could change the default configuration of the simulated processor to see what effects it would have. The code for this project is available at https://github.com/kumj2028/ChampSim.

REFERENCES

[1] Nathan Gober, Gino Chacon, Lei Wang, Paul V Gratz, Daniel A Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition. *arXiv preprint arXiv:2210.14324*, 2022.

[2] Daniel A Jiménez. Fast path-based neural branch prediction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 243–252, 2003.

[3] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.

[4] Gabriel H Loh and Daniel A Jiménez. Reducing the power and complexity of path-based neural branch prediction. In *Proceedings of the 5th Workshop on Complexity Effective Design (WCED5)*, pages 1–8. Citeseer, 2005.

[5] Scott McFarling. Combining branch predictors. Technical report, Citeseer, 1993.

[6] Onur Mutlu. Digital design and computer architecture - lecture 17: Advanced branch prediction (spring 2023). https://www.youtube.com/watch?v=g9H_79ITdbM, 2023.

[7] André Seznec. The o-gehl branch predictor. *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[8] André Seznec. Revisiting the perceptron predictor. *PI-1620, IRISA, May*, 2004.

[9] André Seznec. Analysis of the o-geometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 394–405. IEEE, 2005.

[10] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 8:23, 2006.

[11] Kevin Skadron and David Tarjan. Revisiting the perception predictor again. 2004.

[12] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Washington, DC, USA, 1981. IEEE Computer Society Press.