Branch Prediction Benchmarking

Romario Estrella-Ramos, Ameer Hamza, Jacob Corey

November 2019

1 Introduction

Branch Prediction is a powerful tool in computer architecture that, at best, can greatly cut down on processing time for processors calculating multiple instructions at the same time, such as pipeline or parallel processors. Many branch prediction algorithms have been derived to attempt slight gains of prediction performance, though none are perfect.

For this project, we aimed to test several branch prediction algorithms by using a simulator and sample code to feed branch instructions (among other instructions) to see how well each algorithm performs. Specifically, we aimed to see how each algorithm improved over time as more instructions were executed, as well as see the total sum of both branch prediction 'misses' and 'hits.'

Using the raw data from the simulator chosen, we then attempted to analyze the data and present the results in a meaningful way to show the performance of each branch prediction algorithm in comparison to each other. In this report, the performance of 4 branch prediction algorithms will be analyzed in this manner. Information on the simulator used, how the simulator was used to test branch prediction, how and what data was gathered, and what branch prediction algorithms were benchmarked will be discussed within the paper.

2 Configuration of the Simulator

2.1 The Simulator Chosen

After much research on the best simulator for this topic, we decided upon using the **SimpleScalar** open source computer architecture simulator. The simulator models a virtual computer system with CPU, cache, and Memory Hierarchy. SimpleScalar is capable of simulating Alpa and PISA instruction set architectures. The simulator is execution-driven; it reads a program and simulates the execution of instructions in real time. This is helpful for this experiment as the instructions are processed one-by-one, allowing the simulator to update the status of the micro architecture its simulator and allow us to

record these differences. SimpleScalar is also cycle-accurate as it simulates a microprocessor executing a program on a cycle by cycle basis.

2.2 Modifications Made

The Simulator out of the box gave useful information, such as the amount of branch statements, the amount of hits, and the amount of misses, total. However, we also desired to see a recording of each hit and miss to analyze the data more thoroughly. For this, we modified the source code accordingly in order to obtain the information we desired.

1. Within *bpred.h*, we modified the *bpred_update*() function such that it takes in an additional parameter that specifies the instruction number in which the branch prediciton was performed.

2. Within, *bpred.c*, we modified the same function in such that:

   (a) The new parameter specified previously was also included here

   (b) The function, which is in charge of keeping tally of the branch hits and misses, now outputs a 1 for a hit and a 0 for a miss into a separate text file, $hits.txt$. It also outputs the instruction number of said branch instruction to another text file, $instr.txt$.

3. Within every simulator file under the name of $sim\text{-}*.c$, all instances of a call to *bpred_update*() was changed to pass in the new parameter as well.

With this data, we had chosen a simulator to use and tuned it to give the specific data we required for our own simulation.

3 Branch Prediction Algorithms Tested

For the analysis, we decided to focus on the built-in branch prediction algorithms the simulator supplies: *2-bit bimodal, Two-Level-Adaptive, and a combined branch predictor using the previous two together*. This would make testing simple without having to introduce more branch prediction algorithms manually to the simulator, which would likely include modifying the source code more than what has already been required.

For the code used to analyze branch predictions on, we used three sets of code consisting of the following:

- "Given": Given Sample code that the simulator had available to run.
- "Sample A": Sample C code written by Ameer that tests recursive function calls such as $fibonacchi()$, as well as calls other functions such as $fizzbuzz()$ for various integers. It also included a bubble sort function, a

string concatenation function, and a function to check if a string is a digit, uppercase character, or lowercase character. The idea with this sample code was to thoroughly test branching for function calls. The last mentioned function also used *goto* statements as well, rather than an if or switch statement block.

- "Sample B" Sample C code written by Jacob that focused on creating enormous if/else if statement blocks, as well as enormous switch statement blocks. The if statement blocks test a single *char*'s numerical ASCII value with a possibility for every ascii character. The switch statements originally tested an integer to see if it fell anywhere within 1 to 100000, however, was shortened to only 1000 after filesize issues during simulation. The idea was to test if and switch branching on values that would be hard to predict where they'd branch to.

These sample code collections were then benchmarked with the SimpleScalar simulator by the third member of the group, Romario.

4 Data Obtained from the Simulator

4.1 Raw Data

All 3 test performed decently on all 3 branch prediction algorithms. A summary of collected data involving the total hit/miss count per test below:

	2-Level Adaptive	Bimodal	Combined
Given	6175:1235	6484:926	6586:824
Sample A	123551:6702	123987:6266	125889:4364
Sample B	86234:3138	85920:3452	87320:2052

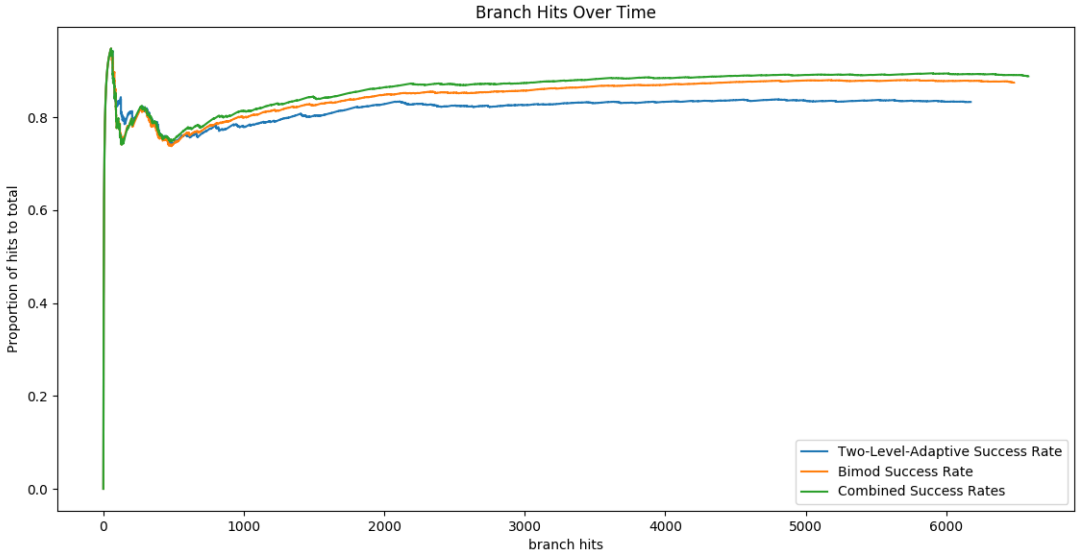
Furthermore, a proportion of hits to total branches can be observed here:

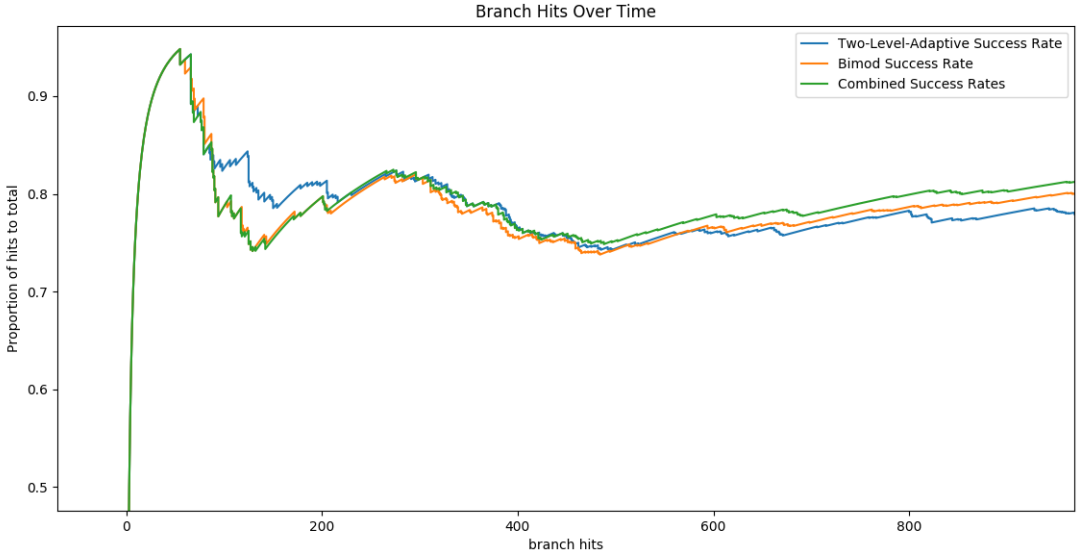
	2-Level Adaptive	Bimodal	Combined
Given	0.833	0.875	0.888
Sample A	0.949	0.952	0.967
Sample B	0.965	0.961	0.977

4.2 Graphed Data

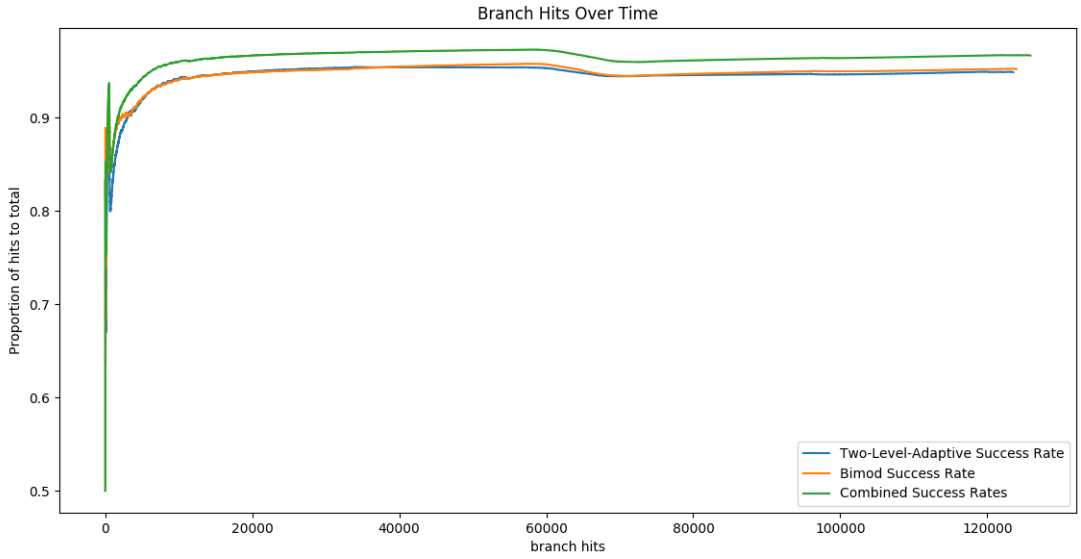
Using the actual record of hits and misses, graphs can be generated for these tests. Both a full graph of the entire test as well as a close-up of the early branch hit trend is given, in order to show both long-term and early trends:

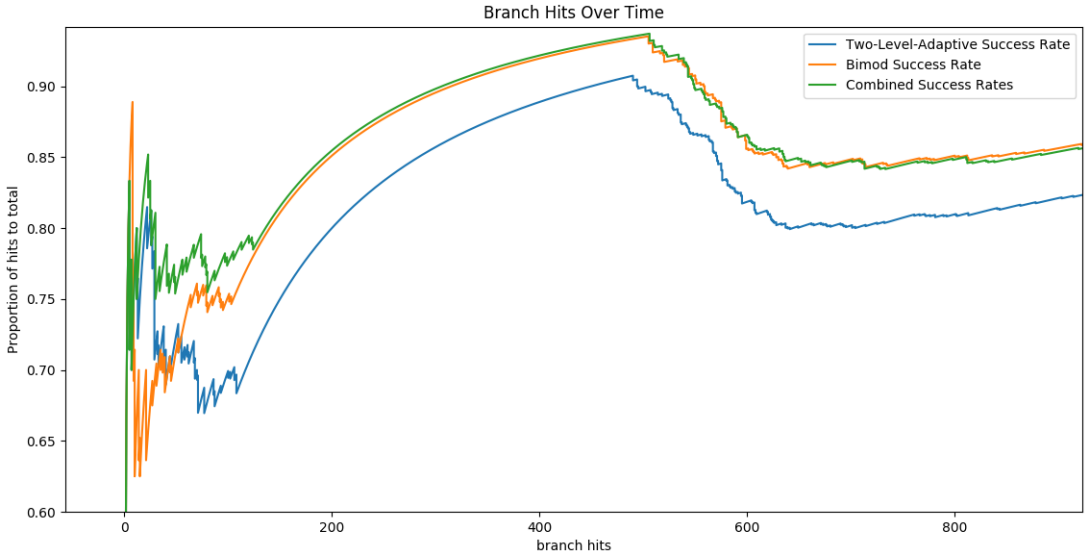
4.2.1 Given


Branch Hits Over Time
Proportion of hits to total
branch hits
Two-Level-Adaptive Success Rate
Bimod Success Rate
Combined Success Rates
0.0
0.2
0.4
0.6
0.8
0
1000
2000
3000
4000
5000
6000

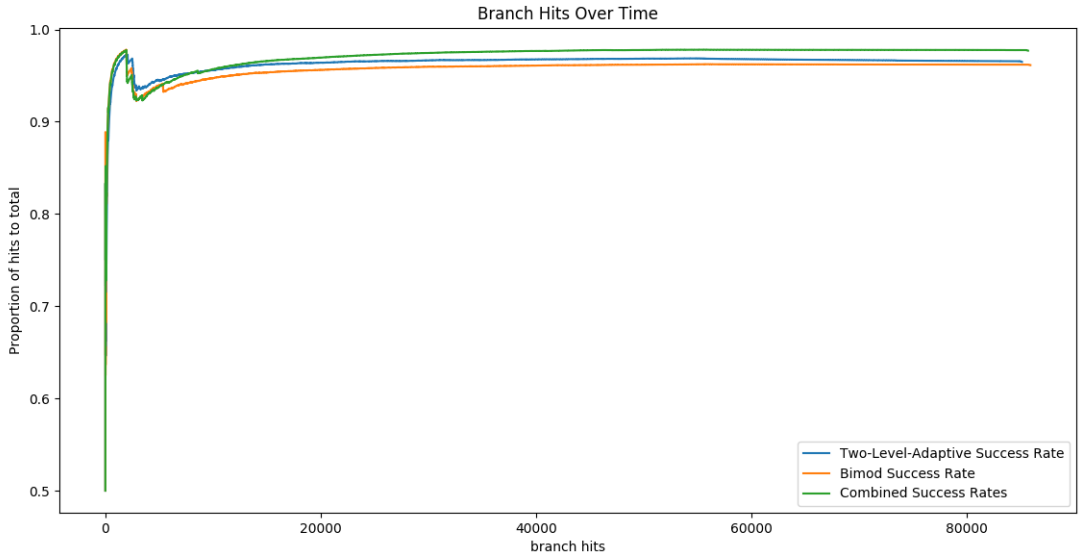

Branch Hits Over Time
Proportion of hits to total
branch hits
Two-Level-Adaptive Success Rate
Bimod Success Rate
Combined Success Rates
0.5
0.6
0.7
0.8
0.9
0
200
400
600
800

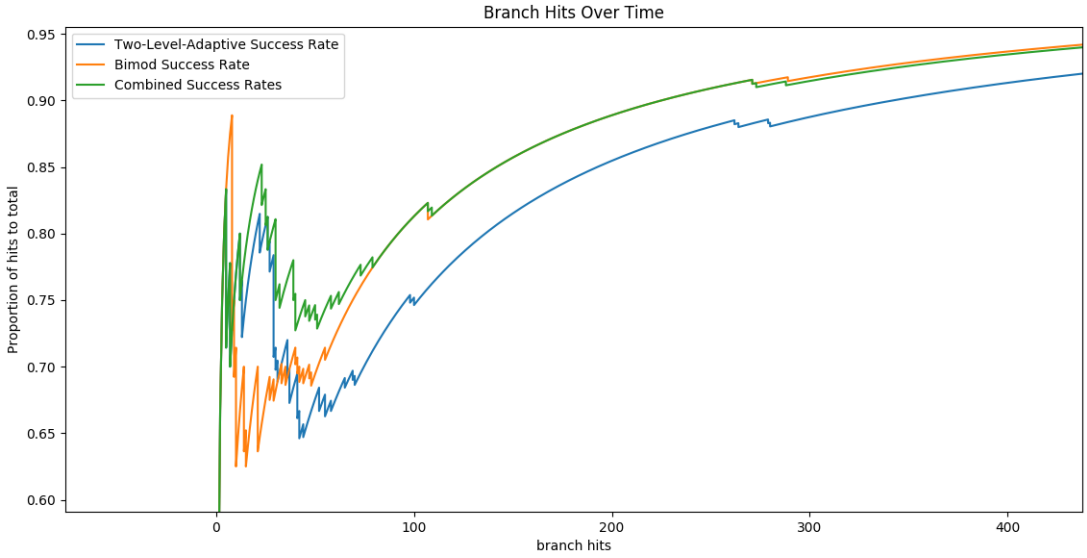
4.2.2 Sample A


Branch Hits Over Time
Proportion of hits to total
branch hits
Two-Level-Adaptive Success Rate
Bimod Success Rate
Combined Success Rates
0.5
0.6
0.7
0.8
0.9
0
20000
40000
60000
80000
100000
120000


Branch Hits Over Time
Proportion of hits to total
branch hits
Two-Level-Adaptive Success Rate
Bimod Success Rate
Combined Success Rates
0.60
0.65
0.70
0.75
0.80
0.85
0.90
0
200
400
600
800

4.2.3 Sample B


Branch Hits Over Time
Proportion of hits to total
branch hits
Two-Level-Adaptive Success Rate
Bimod Success Rate
Combined Success Rates
1.0
0.9
0.8
0.7
0.6
0.5
0
20000
40000
60000
80000


Branch Hits Over Time
Proportion of hits to total
branch hits
Two-Level-Adaptive Success Rate
Bimod Success Rate
Combined Success Rates
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0
100
200
300
400

5 Analysis of Results

While all 3 branch prediction algorithms clearly perform relatively comparably, there are some interesting points of note. First, it's clear that the simulator's *combined* predictor that uses both 2-Level-Adaptive and Bimodal in tandem with each other performs slightly better than either alone. Bimodal tends to perform better than 2-Level-Adaptive, however, when given Sample B which consisted of large amounts of if-statements and switch-case statements, Bimodal did just a little worse than 2-Level-Adaptive. With the Given test code, Bimodal stuck close under Combined in the long run, but actually did the opposite and went lower towards Two-Level-Adaptive's trend-line in the other two simulated pieces of code.

The graphed data also gives us great insight on the early trends, showing that the Combined predictor doesn't necessarily start out as the strogest predictor within the first few hundred branch hits: In Given, Combined briefly dips to the worst performance of the three predictors, before working its way back up. Bimodal takes a very brief early lead in Sample A, before plummeting below the other two as Combined struggles to maintain a stable prediction hit rate at first. In Sample B, despite a slight spike for Bimodal early on, Combined almost always keeps the lead between the three, barely losing it briefly again to Bimodal. Even Two-Level-Adaptive obtains a more successful hit rate very briefly before falling down with Bimodal as the hit rate continues.

6 Conclusion

Of the three branch prediction algorithms tested, the Combined model works the best when compared to either Two-Level-Adaptive or Bimodal, separately. Bimodal tends to perform the next best, unless large if-statements or switch-statements are used, in which case, Two-Level-Adaptive performs just slightly better overall.