Comparison of Branch Predictor Functional Level Models

Parth Saraswat

1. Introduction

As we move towards higher performance processors and techniques such as out-of-order scheduling and superscalar execution, the cost of squashing due to poor branch prediction rises exponentially. This motivates the extreme importance of choosing the best possible branch predictors in order to minimize the cost incurred by poor speculative execution. This project involves implementing different branch predictor schemes, using Pin to evaluate the accuracy of different models, and then evaluating the costs and benefits of each scheme.

To evaluate my design, I utilized a few different frameworks and processes. I used C++ to create functional level models and run simulations for the always taken, 2-bit saturating counter, 1-level BHT, and 2-level parameterized BHT predictors. Then, I utilized Pin to run real workloads on these predictors and collect accuracy data. Pin allows us to do dynamic binary instrumentation on real x86 programs. Therefore, we can use real workloads like gzip to calculate the accuracy of particular prediction schemes.

2. Methodology and Instrumentation

The first approach that I considered when trying to compare branch predictor variations was to create RTL models of different branch predictors and use these models to predict the outcomes of branches of a compiled microbenchmark. However, I immediately realised that this approach did not lend itself to quick and iterative development, and involved excessive overheads in implementation in the shape of creating branch traces before the RTL models can be developed. A much quicker alternative involves instrumenting a real program running natively on an x86 server. Using this method, I could instrument a real x86 program to keep track of the prediction accuracy of an example branch predictor by updating a model of that predictor on every branch. This is the approach that I chose as it allowed us to gain quantitative high-level insight into the performance of each branch predictor variation and identify promising design points, while at the same time creating synthetic microbenchmark traces. Intel has developed a powerful instrumentation tool called Pin which allows us to do all of this.

Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools. Pin provides a rich API that allows us to access context information such as register contents, types of instructions, contents of the instruction pointer, and the outcome of a particular branch instruction. Pin allows us to do this by writing our own 'Pintools' which are C++ programs that describe the behavior of the instrumentation we wish to conduct. In our case, our Pintool was used to instrument binaries of microbenchmarks and benchmarks, and perform 3 basic steps. First, the Pintool would look at each instruction in the compiled binary, and identify branches. Second, for each branch instruction, the Pintool would use a functional level model of a branch predictor to predict the outcome of the branch, and then update the state of the branch predictor depending on whether the branch was actually taken. Finally, as the Pintool has access to the prediction made by the predictor as well as the actual outcome, a helper function also allows it to constantly update the accuracy of the predictor.

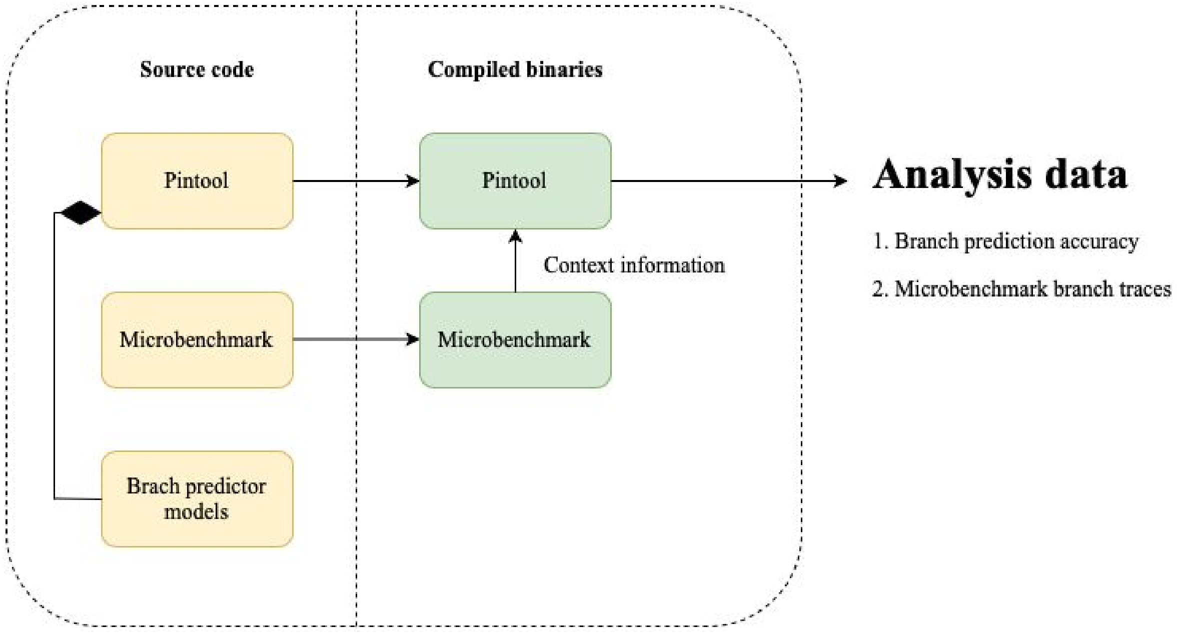


Figure 1: Visual representation of instrumentation methodology
'Context information' includes current instruction pointer, branch outcomes, type of current instruction, etc.

Refer to Figure 1 for a visual representation of this methodology, and Figure 2 for some intuition about how the Pintool functions.

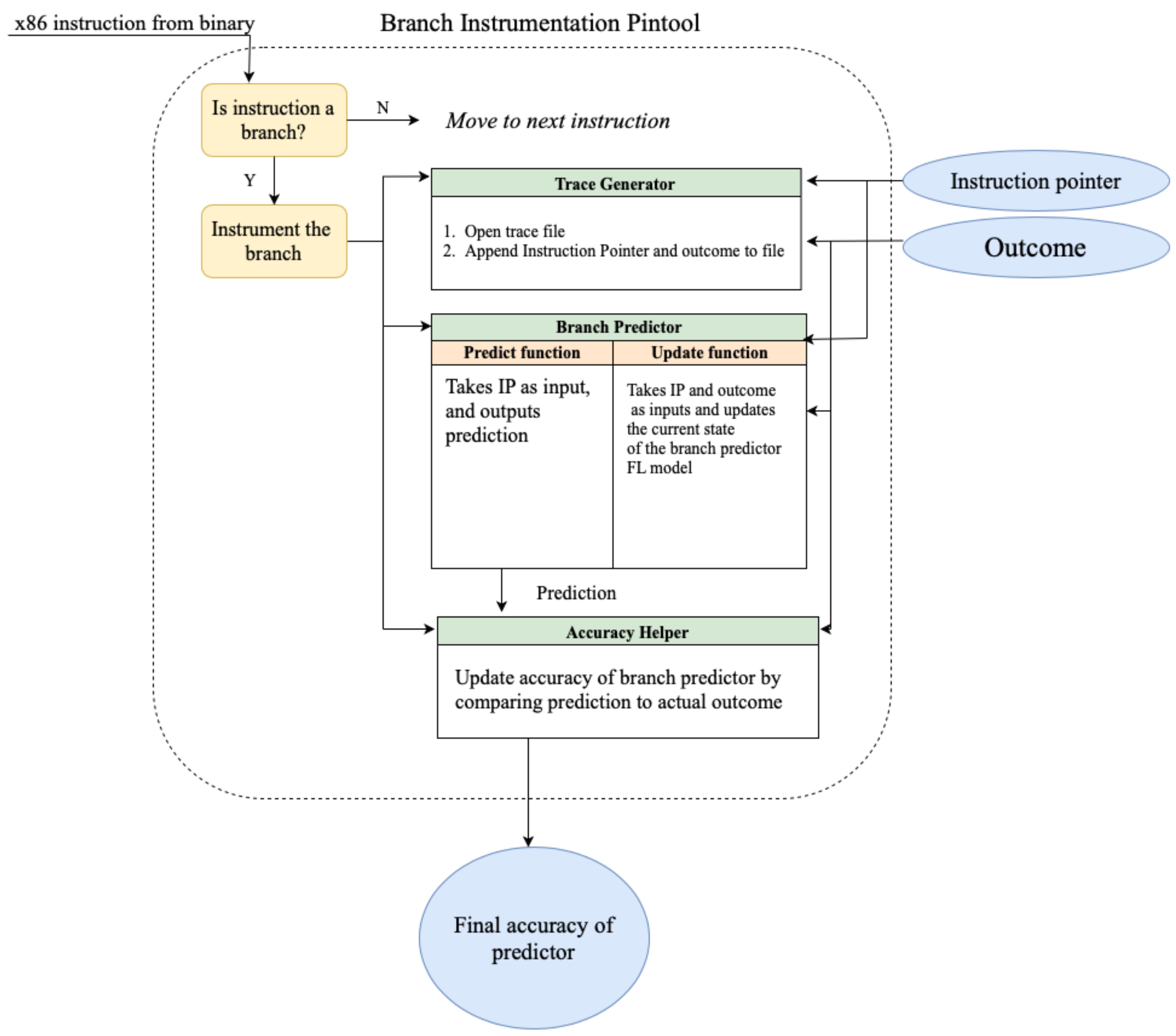

x86 instruction from binary
Branch Instrumentation Pintool
Is instruction a branch?
N
Move to next instruction
Y
Instrument the branch
Trace Generator
1. Open trace file
2. Append Instruction Pointer and outcome to file
Instruction pointer
Outcome
Branch Predictor
Predict function
Update function
Takes IP as input, and outputs prediction
Takes IP and outcome as inputs and updates the current state of the branch predictor FL model
Prediction
Accuracy Helper
Update accuracy of branch predictor by comparing prediction to actual outcome
Final accuracy of predictor

Figure 2: Branch instrumentation Pintool

---

3. Baseline

For the baseline implementation, I used a branch predictor that would always predict that branches were taken.

3.1 Description

This strategy of branch prediction always predicts that a branch will be taken. We chose this strategy because of its ease of implementation, and because of the large room for improvement.

This strategy always makes the same prediction every time a branch instruction is encountered. Because of this, the strategy is termed as a *static* strategy. It has been documented that most unconditional branches are always taken, and loops are terminated with conditional branches which are taken to the top of the loop. This is why we chose to go with a predictor that always predicted taken as opposed to not taken. There is no need to keep track of state as this is a static predictor and does not change its prediction based on the previous outcomes of the branch.

3.2 Implementation

The implementation of this strategy is relatively straightforward. In our Pintool, we set a bool variable called ‘prediction’ to true. The Pintool would then move through each instruction of a compiled binary, and compare its actual outcome to ‘prediction’. The Pintool also has an “Accuracy Helper” function which is constantly comparing the prediction made by our predictor to the actual outcome of the branch, and updating the accuracy of the predictor.

---

4. Testing

Testing this branch predictor is non-intuitive. One methodology is to constantly compare the predictor’s prediction to the actual outcome. The problem with this is that no predictor is 100% correct. Even for a predictor with 99% accuracy, there is one misprediction for every 100 branches. This makes it unclear whether the predictor actually makes a misprediction, or if the functionality of the design is wrong. Therefore, we have to look to other methods in order to be confident in the functionality of our designs. Although we cannot perform black box testing on branch predictors, we can ensure that the predictor is *behaving* as expected. This includes comparing the prediction made by the predictor to the value we expected it to return, as well as whether state updates were made in the right entry of the PHT depending on the contents of the BHSR and the current instruction pointer, and whether the FSM was updated to the right value.

I wrote directed tests to test the behavior of the FL models. These tests included unit tests for the constructors of the predictors, where we checked whether all entries in the predictor were initialised to the correct value, tests for the non-default constructor of the predictors, where we can check whether the number of BHSRs in the BHSRT, the length of each BHSR, as well as the number of PHTs match up with the expected values, a test case for the predict function of the predictors for a given instruction pointer, where we can check whether the output was the same as the expected value, and finally, a test for the update function for a given instruction pointer and outcome, where we can check whether the right entry in the right PHT was updated to the right state. We also wrote an integration test that ran the predictors through a set of predictions and updates, checking whether the entire system worked when integrated together. The FL model implementations also included a few design-for-test functions that were used in order to fetch dimensions and state of particular entries of the predictor. These design-for-test functions allowed for a much more efficient testing process.

---

5. Alternative Design

The alternative designs include multiple functional level models of a variety of dynamic branch prediction schemes. *Dynamic* schemes take into account runtime branch execution history to make predictions. This differs from the *static* scheme described in the baseline section in which prediction is determined from some information that was set before runtime (in our case predicting always taken). The dynamic strategies that that were implemented as part of the alternative design include a 2-bit saturating counter, a parameterised 1-level BHT based scheme, and a parameterised 2-level BHT based scheme. In this section, we will take a close look at the details of each scheme, how we implemented the FL models, some pros and cons of each scheme, and why I chose these designs.

5.1. Two-bit Saturating Counter

The basic building block of any dynamic predictor is the 2-bit saturating counter. This is a FSM that describes the state transitions of the predictor that are used to make predictions. The FSM takes as input the outcome of a branch, and updates its state to reflect what the next prediction should be. Refer to Figure 5 below for the FSM diagram of the 2-bit saturating counter that we used as a building block for all our implementations. When the state is Weakly Taken (WT), or Strongly Taken (ST) the predictor predicts *taken*, and when the state is Weakly Not Taken (WNT), or Strongly Not Taken (SNT) the predictor predicts *not taken.*There are multiple ways that the 2-bit FSM can be modified. For example, the FSM could jump to strongly taken states directly as seen in Figure 6. Another modification may be to change the initial state of the FSM. In our preliminary testing, we observed that the FSM shown in Figure 5 provided us with slightly better accuracy than the other transition logic options, and we also observed that the initial state of the FSM had negligible impact on accuracy, which is why we used the FSM as described in Figure 5.

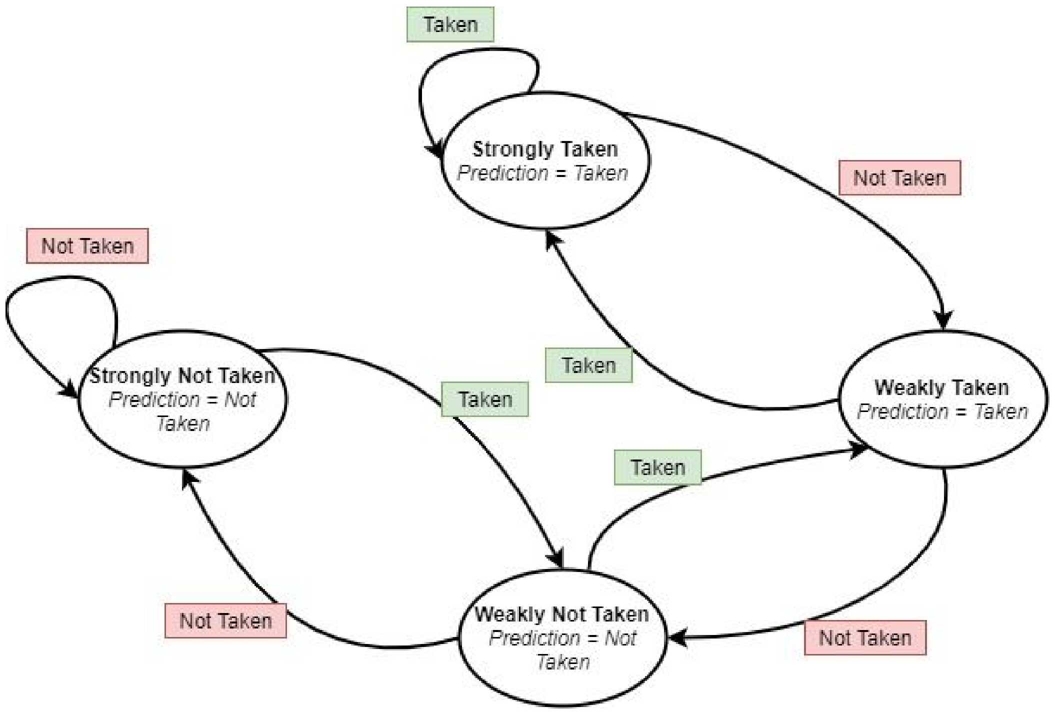


Figure 5: *2-bit saturating counter*

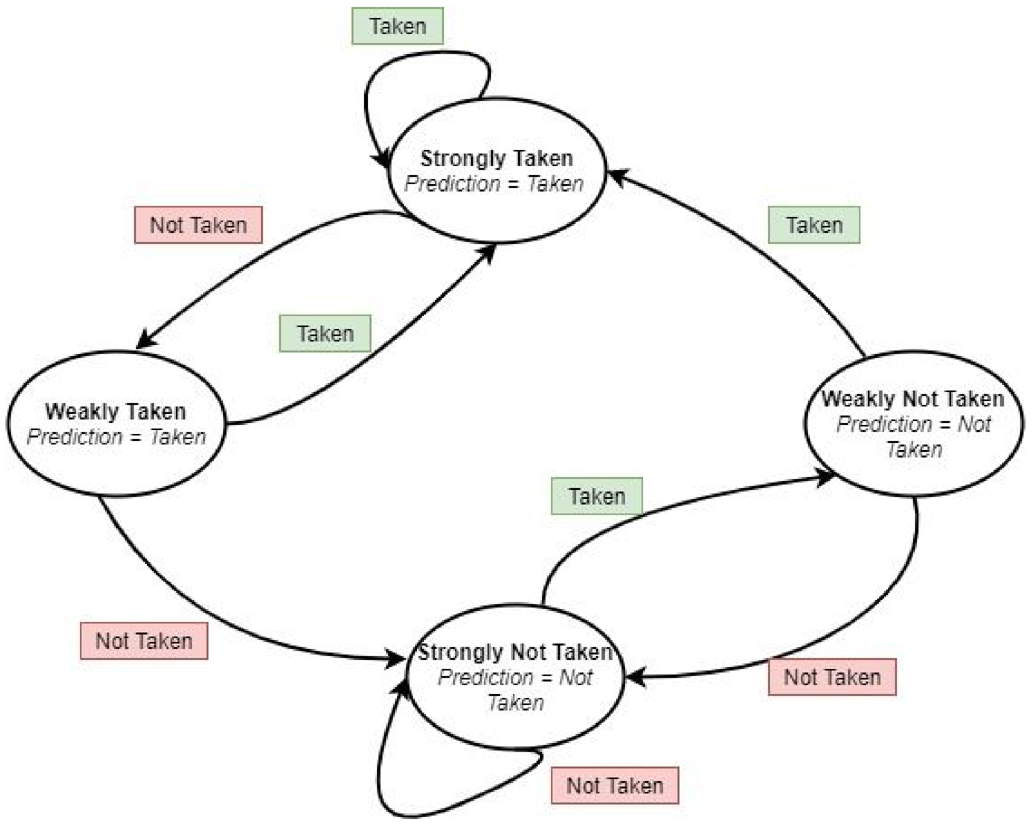


Figure 6: *2-bit saturating counter which jumps directly from weak state to strong state*

I also considered using a 1-bit or a 3-bit saturating counter for our designs, but our preliminary testing showed that the 1-bit design simply did not have enough resolution to capture branch patterns, and the 3-bit saturating counter offered marginal returns. The 2-bit saturating counter offered a good balance in terms of the amount of bits needed, the resolution of the FSM, as well as the ease of implementation. Let's take a closer look at how this was implemented in our FL and RTL models.

All the FL models were built in C++. For the 2-bit counter, the implementation is straightforward. I used case statements to build a simple FSM.

As mentioned previously, the 2-bit saturating counter is incredibly cheap and easy to build. However, this scheme does not offer nearly the amount of accuracy that we are aiming for. The primary reason for this is that each branch in the sequence of instructions is referring to the same FSM, which is causing egregious amounts of undesirable aliasing. In order to remedy this, the next step was to build a predictor that would reduce this aliasing. The way this is done is by implementing a 1-level BHT based scheme, which is why I chose it as the next design.

5.2. 1-Level BHT

The 1-level BHT (branch history table) scheme is nothing but a table of multiple 2-bit saturating counters. This way, every branch does not have to share a single FSM. Now, we can use a certain number of bits from the instruction pointer of the branch instruction to index into this table. The more the number of bits used as the indexing bits, the bigger the BHT will have to be, and the lower the aliasing will be. Refer to Figure 7 for a visual representation of how this scheme works

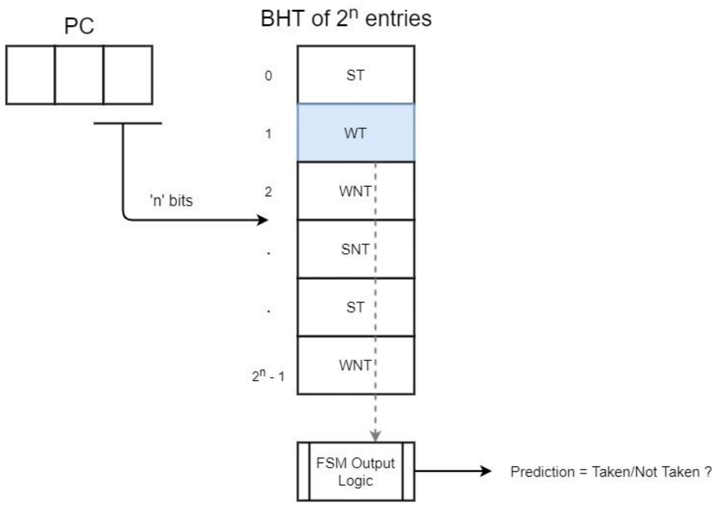


Figure 7: *One-Level BHT Scheme*

Obviously, 2 branches may still alias into the same FSM if they have the same index bits. This problem can be solved by using a bigger BHT. As you may have noticed, the main variable that can be modified in this design is the number of bits used to index into the table (and hence, the size of the BHT as well). In order to study this space, our FL and RTL models of the 1-level BHT are parameterised by the number of bits they use as the index bits. This allowed us to quickly generate 1-level BHT schemes of different sizes and use them to study the effects that larger sizes have on accuracy.

A key feature of this predictor is that it exploits temporal locality. That is, it is capable of detecting the patterns of a particular branch's outcome history, such that the next prediction is dependent on the outcome of the branch. Let's look at how we implemented this predictor's FL and RTL models.

The FL model of the 1-level BHT scheme is a C++ class. The class includes a non-default constructor that can be passed the number of bits you wish to use as the index, and an appropriately sized BHT will be created for you. Other member functions of this class include 'predict' and 'update'. The predict function takes the instruction pointer of a branch as an argument, extracts the index from the pointer using some helper functions, and uses this index to look into the BHT (a vector of 2-bit counters). Finally, the current state of the particular entry of the BHT is used to determine what the prediction is. The 'update' function uses the instruction pointer of the branch and the outcome of the branch (both obtained from Pin as described in section 2) and updates the state of the suitable entry of the BHT. Generating the index from the instruction pointer proved to be a particularly challenging step when trying to implement the FL model. As the instruction pointer generated during instrumentation was a void pointer type, the pointer first had to be cast as a double before we could mask it in order to generate the final index. Casting from a pointer to a non-pointer type is not allowed directly in C++, so we had to move to employing the <reinterpret_cast> ability of C++ in order to work around this issue. The FL models we developed allowed us to quickly get accuracy numbers for different sizes of the 1-level BHT. This helped us in identifying promising design points which we then used when making models that weren't easily parameterised (for example, SRAM sizes).

As mentioned previously, the 1-level BHT is a significant step up from a single 2-bit saturating counter. The main advantages of this scheme is that branches don't alias as much, and temporal locality is exploited when making predictions. However, this scheme is unable to exploit spatial locality. That is, the prediction made for a branch is only affected by its history, and not the history of its neighboring branches. In order to incorporate this global history into our prediction logic, we moved to implement 2-level BHT schemes.

5.3. 2-level BHT

The 2-level BHT scheme is best described as a set of two tables. The first table is the branch history shift register table (BHSRT) and the other table is a set of BHTs. The BHSRT is a table with multiple entries (BHSRs) that keep a track of global history of outcomes, and the collection of BHTs are just a collection of a table of 2-bit saturating counters. We use some bits of the IP to index into the BHSRT (referred to as 'm'), from where we use the value of the BHSRT entry to index into our collection of BHTs. Hence, the length of a BHSRT entry (referred to as 'n') determines how large each BHT in the collection will be. Finally, we use some bits from the IP (referred to as k) to select one of the BHTs from the collection, and use that BHT's output to determine our final prediction. By using this 2-level process, we can now have different predictions for a branch based on the outcome history of neighboring branches. Hence, we are exploiting spatial locality. Now, each entry in the BHT collection maps out to a different *pattern* of a branch's history, which is why the collection of BHTs is actually a collection of *pattern* history tables (PHTs). Refer to Figure 9 for a visual representation of this scheme.

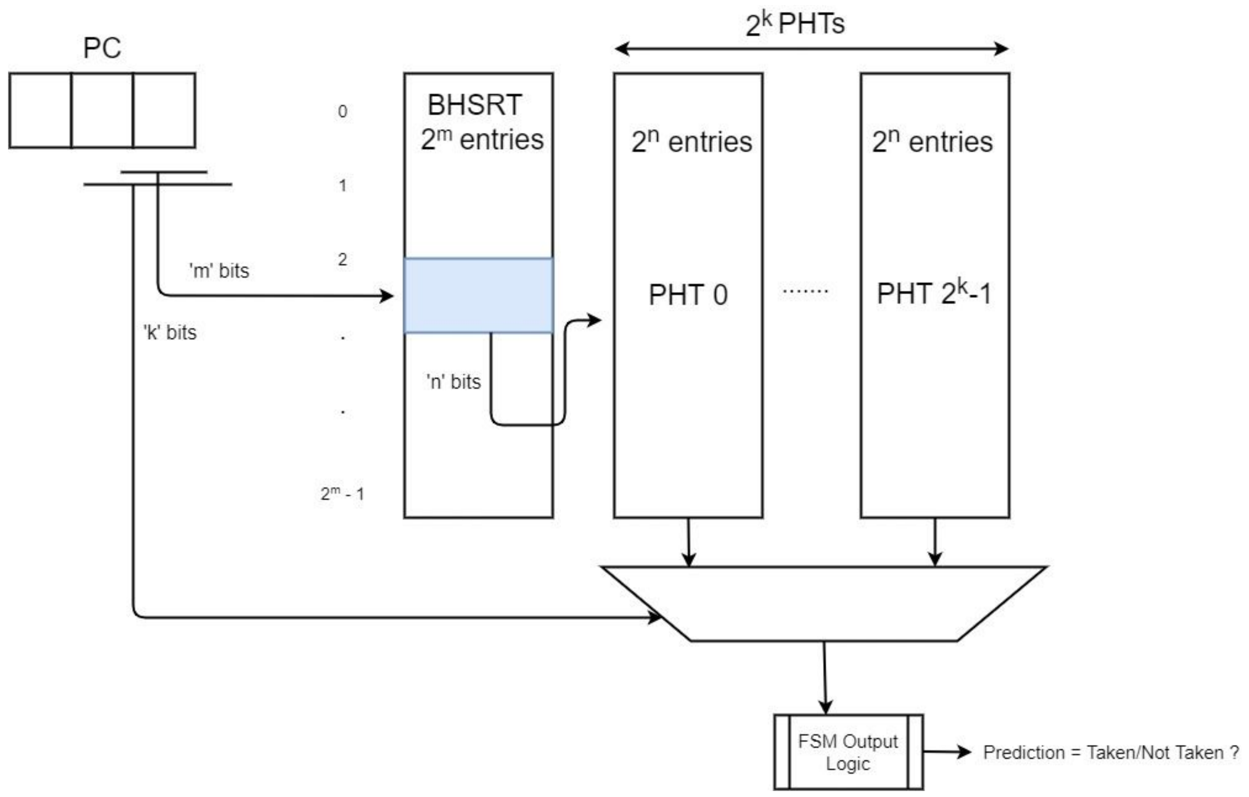


Figure 9: *Two-level BHT*

There are now 3 variables that can be tweaked in order to get different versions of this 2-level predictor (m, n, and k). Our FL model is parameterised by these three values, which allows us to quickly create multiple variations and evaluate their accuracy, and find an optimal combination of these values. The FL model of the 2-level BHT scheme is a C++ class. The class includes a non-default constructor that can be passed m, n and k, and an appropriately sized predictor (a 2-level predictor with 2^m entries in the BHSRT, where each entry is n bits long, and a set of 2^k PHTs where each PHT has 2^n members) will be created. Other member functions of this class include ‘predict’ and ‘update’. The predict and update functions serve the same purpose as the functions in the 1-level BHT. The BHSRT is implemented as a vector of unsigned integers, and the PHT collection is a vector of vectors of states. Having parameterised helper functions really helped us in reusing code, and makes our FL models extensible and modular.

6. Evaluation

We used our Pintool and FL models in combination with a benchmark suite in order to determine the accuracy of each predictor model in the large design space. The benchmarks we used to simulate real workloads were multiple runs of gzip on two different sized files.

All in all, our suite offers applications whose branch counts vary from 46,000 to over 11 million. These large numbers are helpful as a branch predictor tends to have the bulk of its mispredictions in the beginning of a program’s execution, and large branch counts help to mask this effect and allow us to obtain a more representative set of accuracy numbers.

Let’s begin by looking at the performance of our baseline design. The baseline design always predicts taken, and we obtained an average accuracy of **52.243%** on our benchmarks. Using this as a baseline, let’s take a look at how our alternative designs performed.

First, the 2-bit saturating counter. The accuracy results for the 2-bit saturating counter are in Table 2.

2-bit saturating counter - Accuracy results			
	Accuracy		
Memory (bytes)	**gzip (project repo)**	**gzip (small text file)**	**Average**
0.25	47.149616	73.188362	60.168989

Table 2: Accuracy for the 2-bit saturating counter

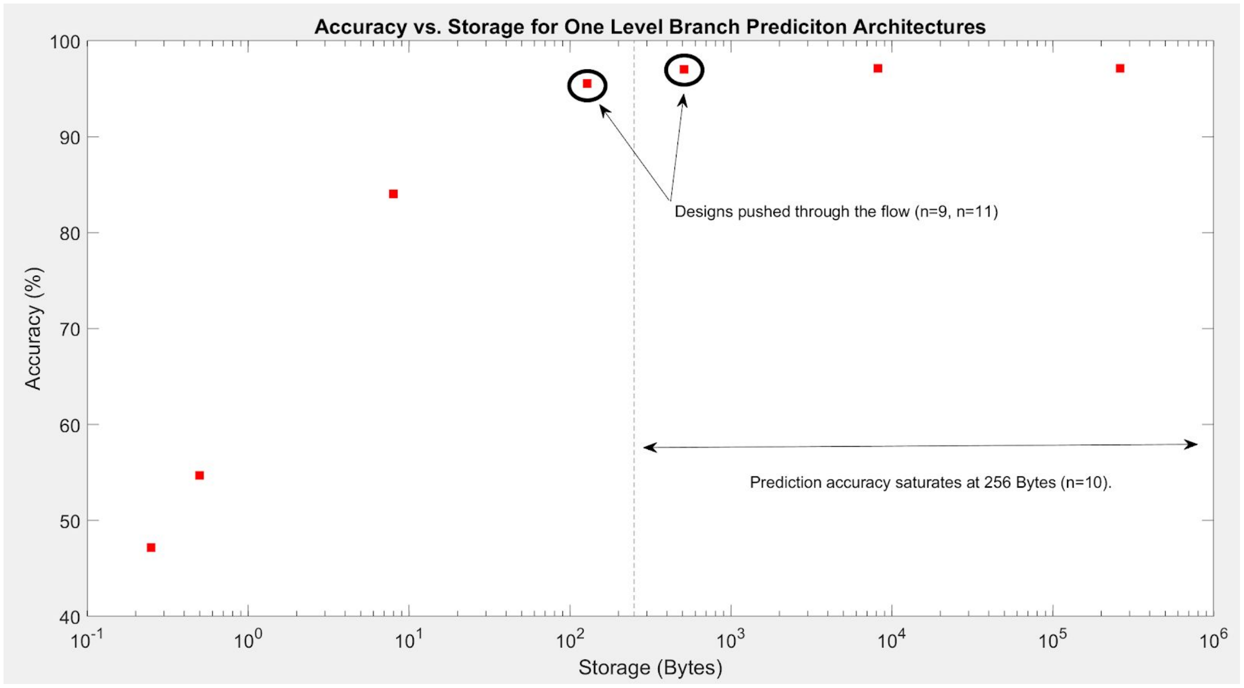
The 2-bit saturating counter has an average accuracy of **60.16%** on our benchmarks. This is a significant improvement over our baseline design, and at the very attractive cost of only 2 bits. It’s interesting to note that the 2-bit saturating counter performed better on the smaller benchmark. In order to determine why, we configured our Pintool to count the number of unique branches in a particular benchmark. This showed that when there was a low number of unique branches, the 2-bit counter performed better. On the other hand, when there are multiple unique branches, the 2-bit counter tends to have lower accuracy than more sophisticated schemes because each branch aliases into the same FSM, and this hurts accuracy. The 1-level design solves this problem. Let’s take a look at the accuracy results for the 1-level design space.

The only variable in a 1-level branch predictor is the number of bits used to index (n) into the BHT (and hence, the size of the BHT). We chose to sweep n from 0 to 20 in steps. The results from this sweep are shown in Table 3 below:

1 -level predictor - accuracy results				
		Accuracy		
n	**Memory (bytes)**	**gzip (project repo)**	**gzip (small file)**	**Average**
0	0.25	47.149616	73.188362	**60.168989**
2	1	59.870171	85.685814	**72.7779925**
5	8	84.090919	92.115959	**88.103439**
10	256	96.781197	95.440712	**96.1109545**
12	1024	97.123497	95.609756	**96.3666265**
15	8192	97.175056	95.613708	**96.394382**
20	262144	97.177574	95.614105	**96.3958395**

Table 3: 1-level predictor accuracy results

This table suggests that as n rises, the accuracy of the 1-level branch predictor rises. This is intuitive, as the size of the BHT grows, aliasing reduces, and accuracy increases. But when does the return in accuracy begin to saturate? Plot 1 below plots the accuracy against the amount of storage needed by a 1-level predictor.

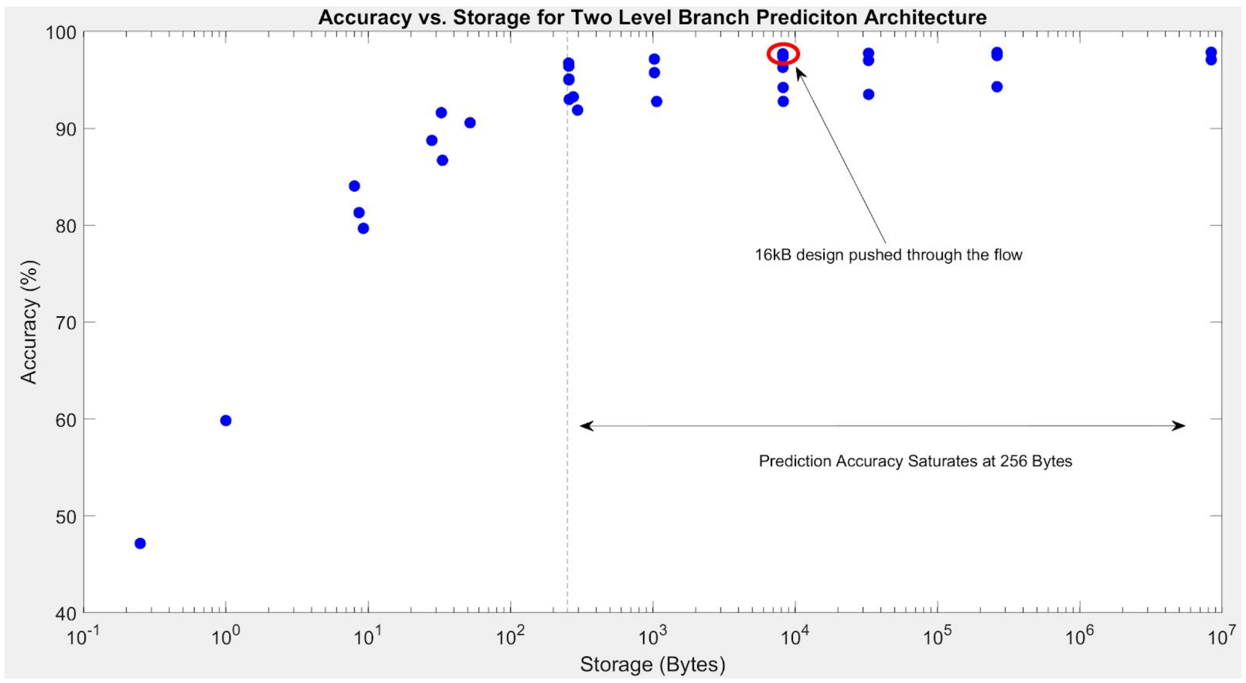


Plot 1: Accuracy vs storage for 1-level predictor variations

When the storage increases from 0.25 bytes (n = 0) to 256 bytes (n = 10), we see a clear increase in the accuracy. Any further increase in storage offers extremely marginal returns. Quantitatively, going from n = 12 to n = 15 offers a **0.02%** increase in prediction accuracy - this corresponds to 1 extra correct prediction every 5000 branches, and it demands **8x** the amount of storage. Hence, it is fair to say that n = 10 and n = 12 are our most promising design points, and we will move forward with implementing their RTL models.

Moving onto the accuracy analysis for 2-level models, the three variables in this case were the number of entries in the BHSRT (dictated by m), the length of each entry in the BHSRT (n bits long) and the total number of PHTs (dictated by k).

We used our Pintool to sweep through 48 possible design points for the 2-level design. The table of results is too big to be included in-line, and can be found in the appendix. In order to determine which designs were to be pushed through the flow, we plotted all of our configurations on a storage v accuracy plot in Plot 2.

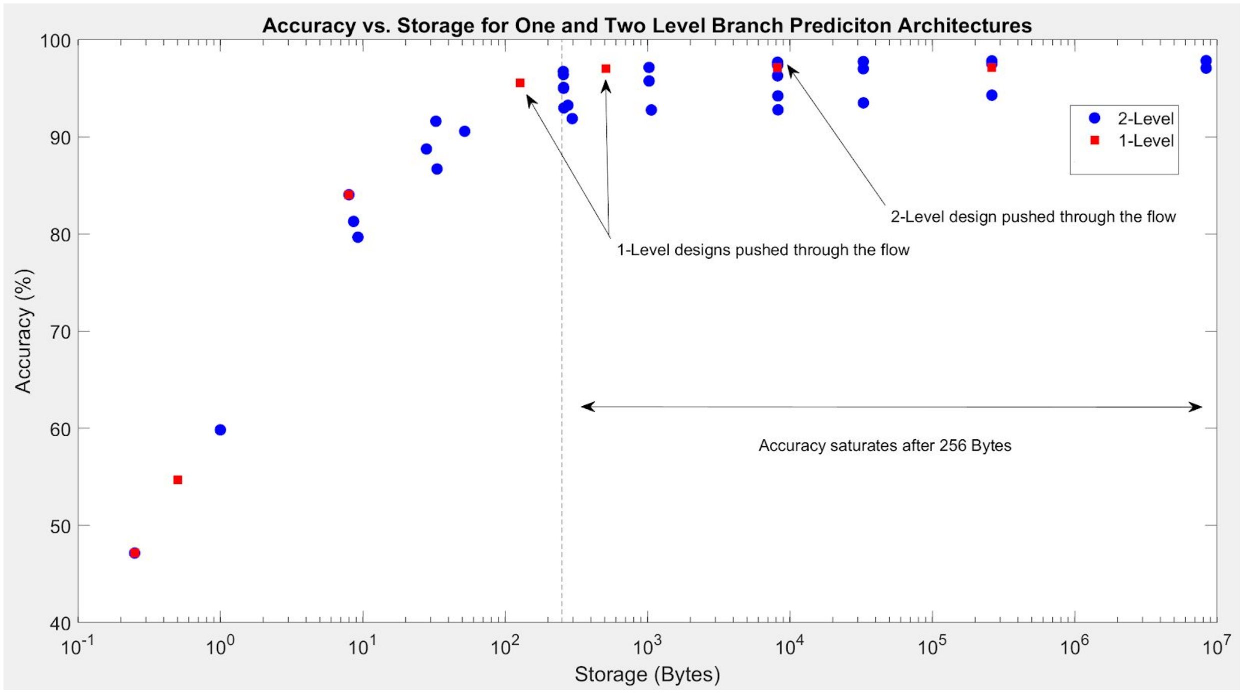


Plot 2: Accuracy vs. Storage for 2-Level predictor configurations

There are multiple Pareto optimal points, but we wanted our 2-level design to outperform the 1-level design, so we picked a 16kB configuration with m = 0, n = 11 and k = 5.

Another key insight is that for 2 designs with the same number of entries in the collection of PHTs (same ‘n’) and the same number of PHTs in the collection (same ‘k’), increasing the number of entries in the BHSRT (m) reduces the accuracy. This lines up with intuition too. As the BHSRT is meant to capture global history, aliasing is desirable. In order to ensure maximum aliasing, we should have the lowest possible number of BHSRT entries. Hence, we settle on m = 0 for our RTL implementation. Knowing this is useful because it allows us to decide early on that we do not need to implement the BHSRT using an SRAM due to its small size.

Plot 3 plots the 1-level and 2-level accuracies on the same plot.



Plot 3: Storage vs Accuracy for all configurations

To summarize our initial sweep on the large design space: we chose n = 10 and n = 12 as promising design points for our 1-level predictor as anything more than that offered marginal returns in accuracy. As for the 2-level predictor, we chose m = 0, n = 11, and k = 6 to be a reasonable design to be pushed through the flow.

Appendix:

Accuracy for different 2-level configurations			
m	**n**	**k**	**Average accuracy**
0	0	0	47.192593
0	0	2	59.461479
0	0	5	83.640381
0	0	10	96.710381
0	5	0	80.927261
0	5	2	91.42437
0	5	5	96.353897
0	5	10	97.673553
0	10	0	94.99408
0	10	2	97.113647
0	10	5	97.640923
0	10	10	97.734329
0	15	0	97.435463
0	15	2	97.73304
0	15	5	97.81427
0	15	10	97.823151
1	0	0	47.192593
1	0	2	59.461479
1	0	5	83.640381
1	0	10	96.710381
1	5	0	79.264687
1	5	2	86.393829
1	5	5	94.929718
1	5	10	97.521095
1	10	0	92.850227
1	10	2	95.691795
1	10	5	97.355309
1	10	10	97.75325
1	15	0	96.253082

1	15	2	96.975052
1	15	5	97.721191
1	15	10	97.831055
5	0	0	47.192593
5	0	2	59.461479
5	0	5	83.640381
5	0	10	96.710381
5	5	0	88.477058
5	5	2	90.320976
5	5	5	93.084175
5	5	10	97.494125
5	10	0	91.675529
5	10	2	92.593956
5	10	5	94.106827
5	10	10	97.478996
5	15	0	92.606361
5	15	2	93.368668
5	15	5	94.16198
5	15	10	97.014526

Table A1: accuracy numbers for all 48 configurations

Annotated Bibliography

[1] James E. Smith. (1981). A Study of Branch Prediction Strategies. *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture,* 135-148. https://doi.org/10.5555/800052.801871

James E. Smith's "A Study of Branch Prediction Strategies" introduces the need of branch prediction and discusses various branch prediction strategies focusing primarily on maximizing prediction accuracy. The paper compares the currently used techniques using instruction trace data and then proposes new techniques to provide more accuracy, less cost and more flexibility. Strategies are divided into two basic categories: (1) static: past history was not used for making a prediction, and (2) dynamic: past history was used for making a prediction. Static branch prediction strategies such as predicting that all branches will always be taken/not taken are inexpensive and simple to implement and provide a good standard for judging other branch prediction strategies. Other static branch prediction strategies that exploit the program execution structure are opcode-based prediction and branch-based prediction i.e. based on forward or backward branches. The paper also lists down dynamic strategies that improve accuracy and are more physically realizable.

[2] Scott McFarling. (1993). Combining Branch Predictors. *Western Research Laboratory Technical Note TN-36.*

Scott McFarling's "Combining Branch Predictors" describes multiple dynamic prediction schemes with an eventual goal of combining multiple predictors in order to gain the maximum possible accuracy. The different dynamic prediction schemes include bimodal predictors, local predictors, and global predictors. The paper also proposes a method to combine the best features of these predictors into a combined design, but our focus will be on the individual schemes themselves. We will be picking one of the individual schemes based on Pin evaluation and implementing it as a part of our alternative design. This will provide us with another design point to consider when evaluating trade-offs.

[3] Tse-Yu Yeh, & Yale N. Patt. (1991). Two-Level Adaptive Training Branch Prediction. *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture,* 51-61. https://doi.org/10.1145/123465.123475

The authors, researchers at the University of Michigan in 1991 when this paper was published, examine several previously implemented branch prediction schemes, both static and dynamic, as well as propose their own scheme - the Two Level Adaptive Training Scheme. They detail the scheme at a high level, and then dive into different ways to implement the required data structures. The authors then give a detailed

comparative analysis of the performance of their own scheme compared to previously implemented schemes. They also do an analysis on the various proposed implementations of the Two Level Adaptive Training Scheme, and settle on an ideal implementation.
This paper provides insight into the motivation, design, implementation, simulation, and evaluation of our proposed alternative design. While the functionality of the design that the authors outline is useful, the most relevant part of the paper is the detailed accounts of their implementations, as well how they simulated their predictor. This will become very important as we try to navigate implementing and simulating our own predictor.