Comparison of Branch Predictor Variations

Jake O'Brien (jro77), Parth Saraswat (ps978), Shivangi Gambhir (sg2439)

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. The goal of the project is to provide a conclusion of which branch predictor scheme to choose based on particular constraints. Current literature evaluates branch predictors solely using accuracy as a metric. We evaluate our designs across metrics like accuracy, area, energy, and cycle time, therefore better understanding the tradeoffs if any between accuracy and area, energy and cycle time.

In the current literature we found three papers which gave us significant background on different branch predictor schemes and strategies. The papers also included designs of reasonable scope for this project. These papers were James E. Smith's "A Study of Branch Predictor Strategies", Tse-Yu Yeh, & Yale N. Patt's "Two-Level Adaptive Training Branch Prediction" and Scott McFarling's "Combining Branch Predictors". Smith's paper introduces the need of branch prediction and discusses various branch prediction strategies focusing primarily on maximizing prediction accuracy. Tse-Yu Yeh, & Yale N. Patt's "Two-Level Adaptive Training Branch Prediction" builds upon Smith's paper by examining several previously implemented branch prediction schemes, both static and dynamic, as well as proposing their own scheme - the Two Level Adaptive Training Scheme. Finally, Scott McFarling's "Combining Branch Predictors" builds upon the first two papers by describing multiple dynamic prediction schemes with an eventual goal of combining multiple predictors in order to gain the maximum possible accuracy.

The baseline design for this project is the functional level model of a static scheme described in "A Study of Branch Predictor Strategies", in which branches are predicted as always taken. The static, always taken scheme is a good baseline design due to its simplicity, and basis of comparison for other designs. We implement prediction of always taken over always not taken because a majority of branches are usually taken, and thus we should expect higher accuracy than if we implemented always not taken[1]. In addition, this design has high program sensitivity and thus it is a good point of comparison against our alternative design because we get a good sense for the improvement in program sensitivity in our alternative design.

For the alternative portion of the project, we implement the functional level models for a 2-bit saturating counter, a parameterized 1-level BHT which keeps track of local history, and a parametrized 2-level BHT which keeps track of local and global history in Pin. We also then implement an RTL model for parameterized 1-level and 2-level BHTs. These predictors are dynamic prediction schemes adopted from Tse-Yu Yeh and Yale N Patt's Two-Level Adaptive Training branch predictor. This predictor uses two levels of branch history information in order to make the prediction. The first level is the history of the last 'n' branches, and the second level captures the branch behavior for the last 's' occurrences of the unique pattern of the last 'n' branches. A branch that is taken is given a "1", and a branch that is not taken is given a "0". The current history in the first level is used to index into the second level. A FSM in the second level is used to yield a prediction for that unique pattern of branches. This scheme allows for high accuracy over many different programs and data sets ["Two-Level Adaptive Training Branch Prediction"]. Results from the functional level model of the 2-level predictor are used to tune the RTL model. Further, the other alternative designs implemented solely in Pin serve to give a comparison point of accuracy versus the two-level predictor. The RTL design that gets pushed through the ASIC flow allows us to understand the design space in depth for the two-level predictor. In addition, as a reach goal we look at the process implemented by "Combining Branch Predictors" in which the best parts of several designs are combined.

To evaluate our design, we utilized a few different frameworks and processes. We first 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. We 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. The existing literature provides extensive comparison using accuracy as their main metric. Therefore, after we collected accuracy information using Pin, we implemented synthesizable RTL models for promising designs, and quantitatively evaluated the alternative designs on metrics such as area, energy, and cycle time. This gave us a broader idea of the design space that branch predictors occupy, and better allowed us to understand the tradeoffs of accuracy against area, energy, and cycle time.

The progression of the literature that we referenced follows the progression of our design approach. They start off with simpler designs and build off of each other over time with the goal of creating more accurate branch predictors. This makes sense if we look at the date that these papers were published. A Study of Branch Predictor Strategies was published in 1981, Two-Level Adaptive Training Branch Prediction in 1991, and Combining Branch Predictors in 1993. The evolution of branch prediction is incremental over time. This is a logical design approach, and we did our best to emulate this within our own project.

A common theme throughout the literature is the need for branch prediction. Smith writes, "in high-performance computer systems, performance losses due to conditional branch instructions can be minimized by predicting a branch outcome." Published nearly 40 years ago, it

is clear that as the performance of computer systems continues to increase, the need for highly accurate branch prediction schemes is more necessary than ever.

Where the papers differ is their proposed schemes to improve accuracy, as well as their methods for testing/evaluating the implemented predictors. Smith's 1981 paper spends a good deal of time on static predictor schemes. These are schemes in which past history is not used for making a prediction. Several different static schemes are highlighted such as predictions based on opcode, and predictions based on whether the branch is a forward or backward branch. However, we focused on the scheme in which all branches are predicted taken, because that is our proposed baseline design. The scheme is run on six FORTRAN programs. Based on this scheme the best performance yielded 99.4% accuracy, and the worst performance yielded a 57.4% accuracy ["A Study of Branch Prediction Strategies"]. The takeaway here is the program sensitivity to branch prediction accuracy - a point we explored within our own baseline design and used as a basis of comparison. The 1981 paper also introduced some dynamic prediction schemes such as only predicting not taken if a branch is found in a history table containing previous branches that were not taken, or making a prediction based on a history bit. The Two-Level Adaptive Training Branch paper builds off the dynamic schemes described in "A Study of Branch Predictor Strategies" by describing a two-level adaptive predictor scheme which we use for our alternative design. Nine benchmarks from the SPEC benchmark suite are used in the study, and instruction traces are fed to the branch prediction simulator to collect statistics. The proposed scheme was able to get an average accuracy of 97% on the benchmarks ["Two-Level Adaptive Training Branch Prediction"]. Finally, "Combining Branch Predictors" builds off the first two papers by describing more dynamic prediction schemes, and how to combine the best parts of these designs into one combined design. The combined design can yield an accuracy of 98.1% ("Combining Branch Predictors"). A similarity in this paper and "Two-Level Adaptive Training Branch Prediction" is that they both use the SPEC 89 benchmark suite on their schemes.

We evaluated our designs on the following metrics: accuracy, area, energy, cycle time and performance. We realised that the accuracy of the implementations begins to saturate at 256 bytes. A key theme of the entire comparison was the tension between regfile and SRAM implementations of the 1-level architecture. The regfile implementations were smaller for lower storage, but for ideal sizes with good accuracy, the SRAM implementations were far more area and energy efficient. The regfile implementations were also able to perform predictions and updates in one cycle, whereas the SRAM implementations needed 3 cycles. This caused the regfile implementations to have much better performance. In terms of cycle time, we realised that a large 16kB 2-level predictor cannot be implemented using a single SRAM, but needs to employ a banked architecture in order to keep cycle time reasonable.

2. Methodology and Instrumentation

The first approach that we 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, we 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, we 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 we 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.

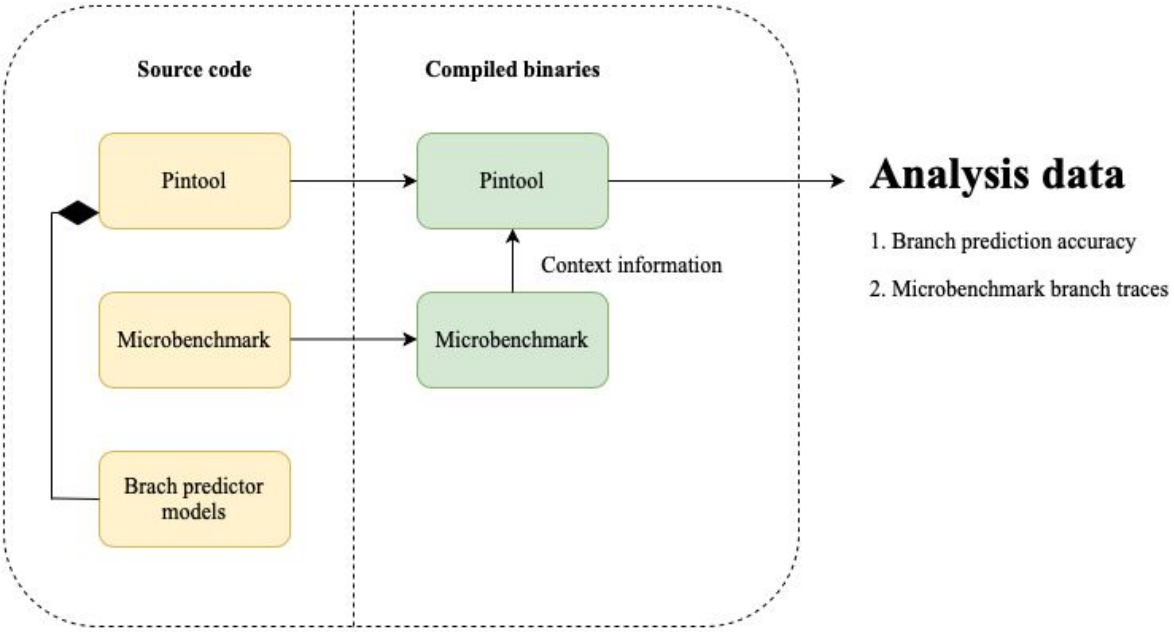


Figure 1: Visual representation of instrumentation methodology

'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.

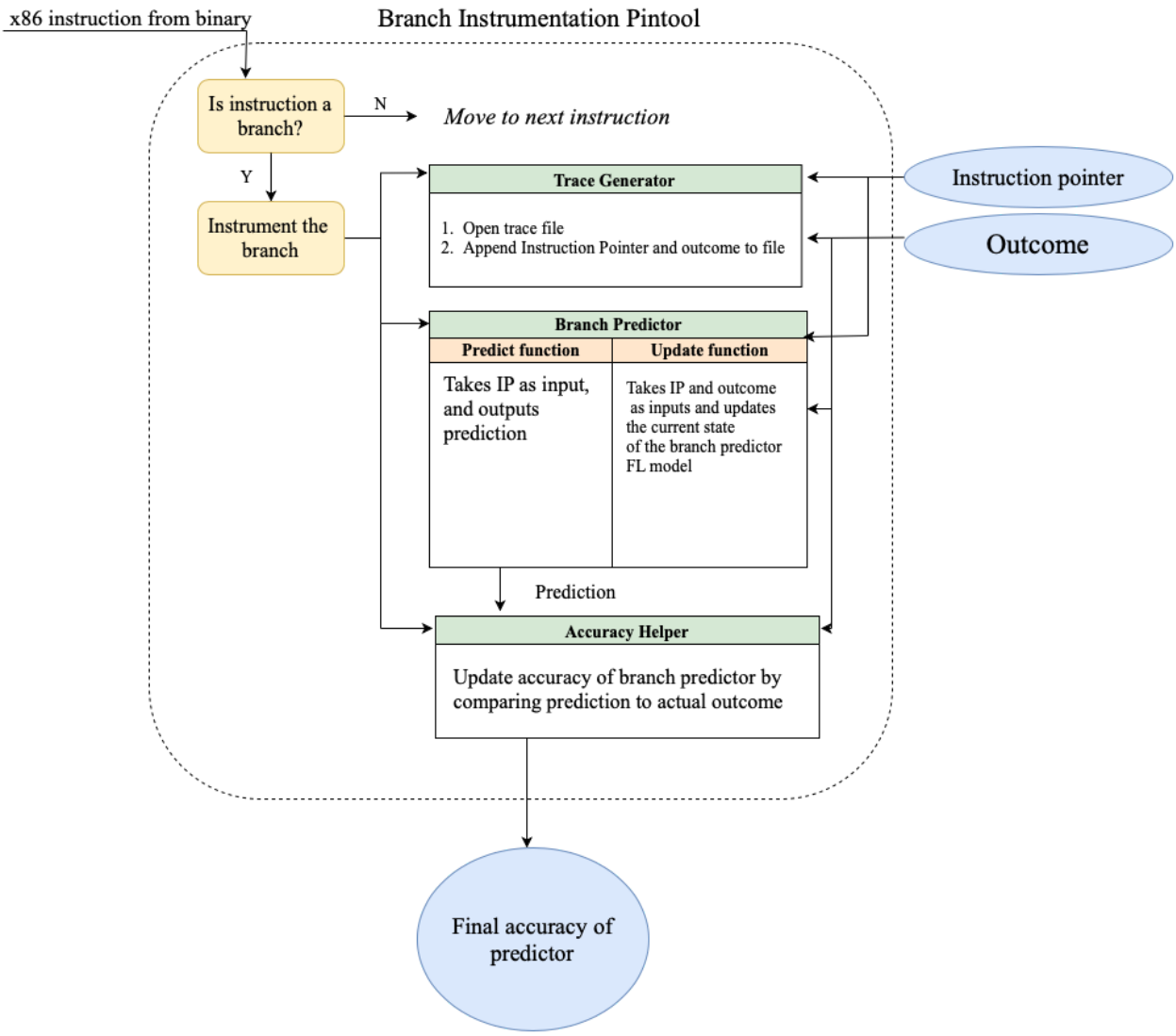


Figure 2: Branch instrumentation Pintool

---

3. Baseline

For the baseline implementation, we 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[1]. 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[1]. 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.

We wrote directed tests to test the behavior of our 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.

To test our RTL models, we utilized two testing strategies. The first strategy used directed testing to make sure the RTL was functioning correctly, and the second strategy compared the accuracies resulting from the Pin tool and from a simulator that we created.

The first part of our RTL testing strategy used directed testing. Separate directed tests were used to test the one-level, and two-level designs. To test the one-level design, we stressed indexing into the same entry in the BHT, as well as indexing into multiple entries in the BHT. When indexing into multiple entries of the BHT, the directed tests looked at accessing the same entry multiple times, followed by another entry multiple times, as well as when entry accesses were interleaved. Each directed test worked its way around the FSM, making sure to make the anticipated predictions for every FSM state. To test the two-level design, we first wrote a simple test case to test if multiple branch patterns could be written and read correctly from a single PHT. We then tested to ensure the BHSRT was functioning properly and updating with the right outcomes of previous branches. This involved hopping from one IP to another to see if the BHSRT was updated properly. Finally, we used multiple branches to hop from one PHT to another PHT to make sure there was no conflict between the PHTs. We also made sure that the SRAM got updated correctly, and that one PHT wasn't overwriting the contents of another.

The second part of our RTL testing strategy looked at comparing the accuracies produced from the pin tool to those from the RTL implementations. We first used Pin to create traces of microbenchmarks, and then used those traces as inputs to our RTL models. To verify correctness, the accuracies between the FL models in Pin and the RTL models should match up exactly. Table 1 shows a comparison between the FL and 1-level register file RTL implementation accuracies on five microbenchmarks using 9 and 11 index bits. The simulator that produced the accuracy measurements works by parsing through a text file of branch traces, instantiating one of the RTL models, and feeding the inputs from the trace into the model. The simulator is then able to compare an array of predictions made from our model with the outcomes from the branches given in the traces, and thus determine accuracy. An example of the output from running the simulator is given in Figure 3, and an example trace is shown in Figure 4. This trace gives the instruction pointer as well as the outcome for each branch. The reason why the accuracies don't noticeably change between the n=9 and n=11 index bits is because the trace's IPs that were used don't change between the 9th and 11th bits. It is also important to note

that the slight accuracy differences seen between the FL and RTL models is due to differences in floating within Python and C++.

FL Model accuracy for ubmarks					
n	**accum**	**bsearch**	**cmult**	**sort**	**vvadd**
9	87.096771	85.486206	86.206894	76.314453	86.206894
11	87.096771	85.486206	86.206894	76.365494	86.206894
RTL Model accuracy from simulator for ubmarks (Regfile)					
n	**accum**	**bsearch**	**cmult**	**sort**	**vvadd**
9	87.096774	85.486211	86.206896	76.314446	86.2068965
11	87.096774	85.486211	86.206896	76.365492	86.2068965

Table 1: Accuracy Results for the FL and Simulator

```
 1    IP: 0x400702 O: 0
 2    IP: 0x400738 O: 0
 3    IP: 0x400741 O: 0
 4    IP: 0x400775 O: 1
 5    IP: 0x400775 O: 1
 6    IP: 0x400775 O: 1
 7    IP: 0x400775 O: 1
 8    IP: 0x400775 O: 1
 9    IP: 0x400775 O: 1
10    IP: 0x400775 O: 1
```

Figure 3: First 10 branches in vvadd ubenchmark trace

```
num_cycles      = 691
num_branches    = 689
num_correct     = 589
accuracy        = 0.8548621190130624
```

Figure 4: Example output from the simulator for 1-level regfile using bsearch ubenchmark

5. Alternative Design

Our alternative designs include multiple functional level and register-transfer level models of a variety of dynamic branch prediction schemes. *Dynamic* schemes take into account runtime branch execution history to make predictions[2]. 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 we implemented as part of our 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 and RTL models, some pros and cons of each scheme, and why we 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.

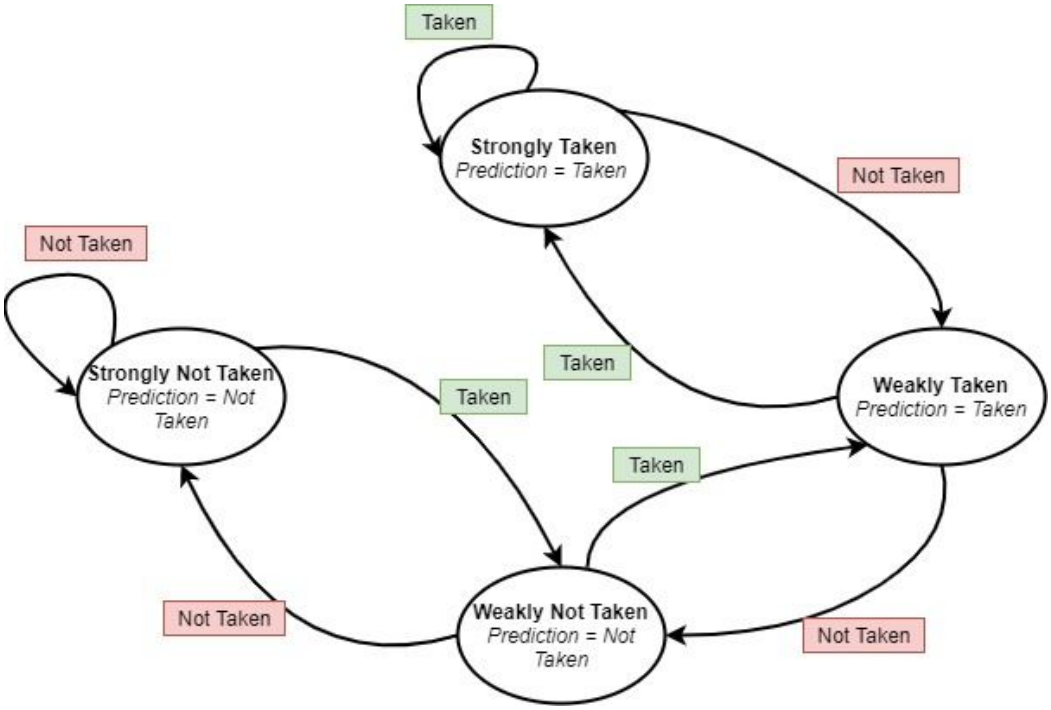


Figure 5: *2-bit saturating counter*

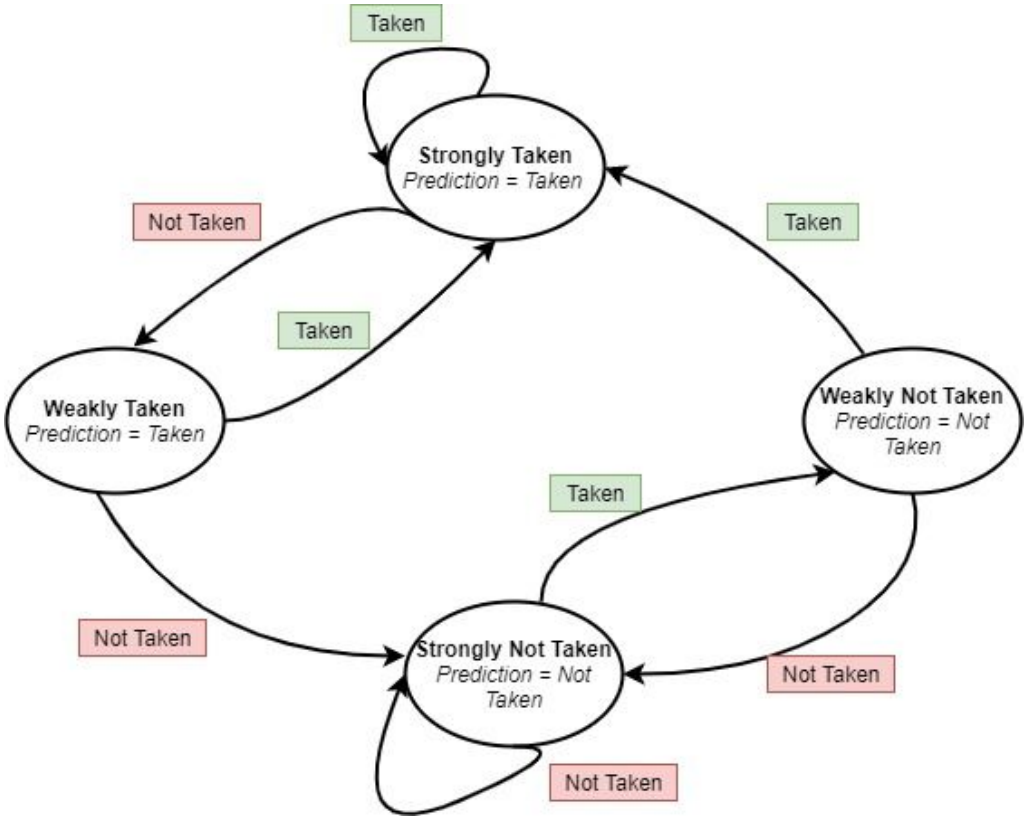


Figure 6: *2-bit saturating counter which jumps directly from weak state to strong state*

We 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 our FL models were built in C++, and all our RTL models were written in Verilog. For the 2-bit counter, the implementation is straightforward. We used case statements to build a simple FSM for our FL and RTL implementations.

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, our 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 we chose it as our 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

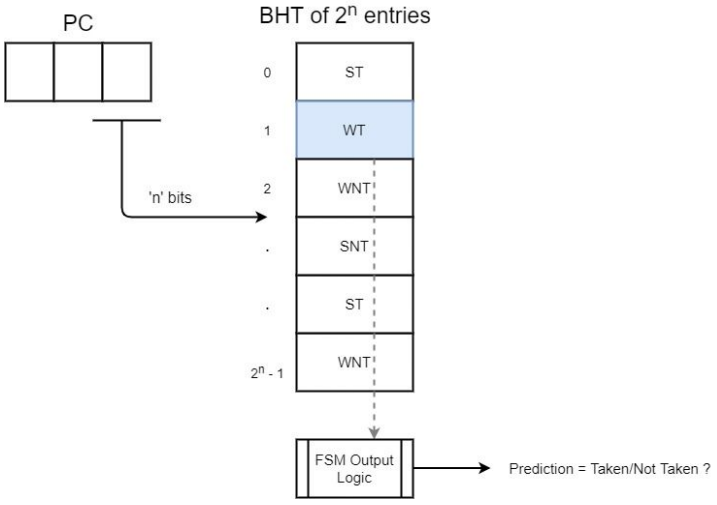


Figure 7: *One-Level BHT Scheme*

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).

The RTL model of the 1-level BHT was implemented using a register file, as well as by using SRAMs. We will first examine the 1-level BHT implementation using a register file. Similar to the FL model, the RTL model also supports the 'predict' and 'update' functionality. Its interface takes as input an instruction pointer, and the branch outcome, and outputs the branch prediction. A key feature of the register file design is that it is able to perform a predict and update in a single cycle. We first implemented this design in a way that took multiple cycles, but eventually managed to condense the update and predict functionality into a single stage as described above. This was important, as taking less cycles is the advantage of a register file implementation over an SRAM implementation.

The RTL model of the 1-level BHT with SRAMs is similar to the RTL version of the register file implementation, except now, the BHT is implemented as an SRAM. It also performs

predictions and updates similarly. However, as reads and writes from and to the SRAM aren't combinational, they require extra cycles. This means that a particular transaction (predict + update) is only completed after three cycles. This functionality was implemented by the use of an FSM which can be seen in Figure 8, and also required the use of an en/rdy handshake protocol, as the predictor would not be ready for new inputs every cycle (unlike the register file implementation). The SRAM version of the 1-level BHT sends a read request to the BHT in the WAIT state. The response is received in the PREDICT state, the prediction is made, and marked as valid. The next state is also calculated in this stage. Finally, a write request is sent to the SRAM in the UPDATE state. All in all, the SRAM implementation takes 3 cycles to complete one transaction.

A key challenge when incorporating SRAMs was the problem of partial writes. As we only need to update 2 bits of a particular row of the SRAM, we had to register the line that was read out from the SRAM, calculate the 2 bits that needed to be updated, and then write the entire line back into the SRAM. Had we been able to perform partial writes by default, the predict and update transactions would have taken 2 cycles instead of 3. The reason behind using an SRAM is simply that as the number of entries in the BHT are increased for better accuracy and lower aliasing, the size of BHT increases exponentially and SRAMs become more suitable as they provide a more compact and dense design.

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.

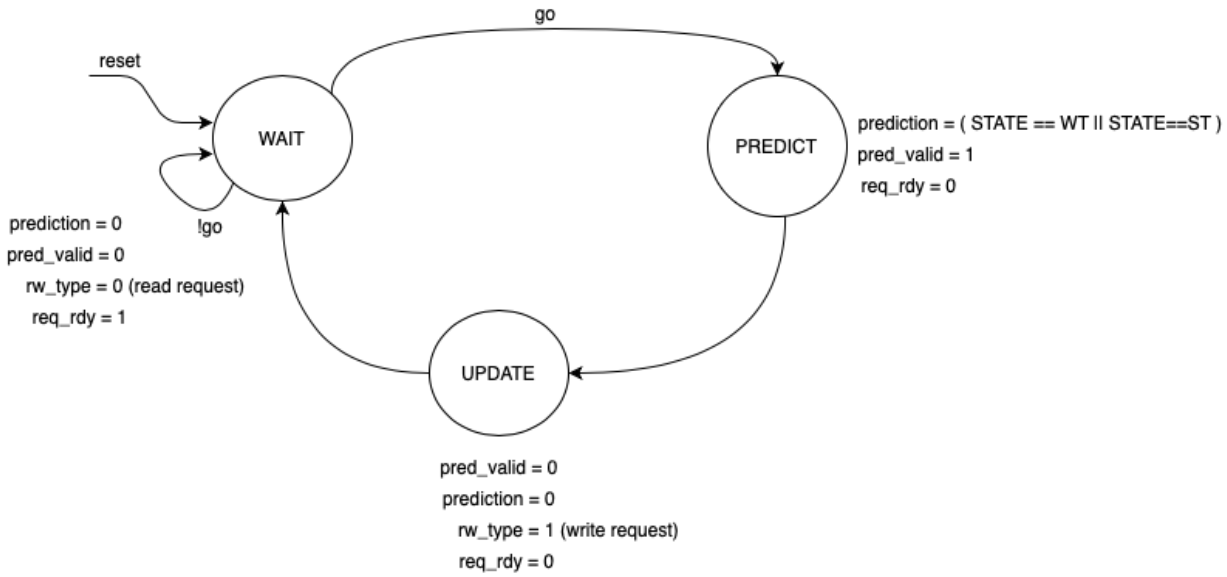


Figure 8: *FSM used in the SRAM RTL implementation of the 1-level and 2-level Predictor*

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.

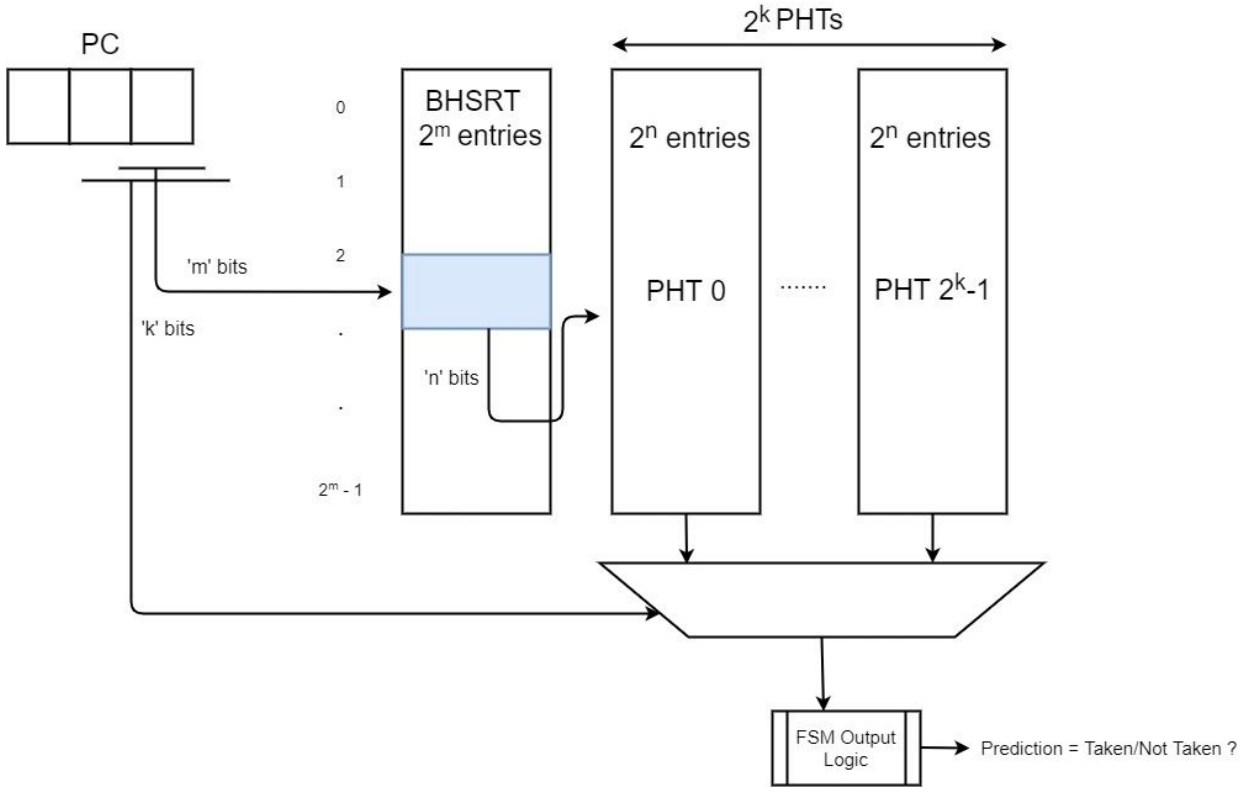


Figure 9: *Two-level BHT*

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.

The high level design of the RTL follows that of the FL. We parameterize m, n, and k such that we can make different 2-level BHTs. This will act as a 'branch predictor generator' of sorts. The RTL model of the 2-level predictor follows the same FSM as the 1-level predictor with three states: WAIT, PREDICT and UPDATE as seen in Figure 8. While exploring the design space, we came up with two implementations of the SRAM for the most promising numbers we obtained using the functional level models. We initially started with implementing a 64x2k SRAM which turned out to be expensive in terms of cycle time which will be explored further in the next section. To reduce the cycle time, we implemented a multi-bank SRAM where the 64x2k SRAM was split into multiple smaller SRAMs.

While there are several ways to configure a multi-banked implementation, we settled on four 64x512 SRAMs. Although there may be small area improvements from other configurations, this design was chosen based on implementation complexity considerations. The 64x2k block was broken to four 64x512 banks as seen in Figure 10. This allows the number of columns to remain the same which allows us to reuse our logic to select specific data bits. The only logic which changes is how we choose the index or the row, and this is fairly easy to implement. Each bank has its own 'val' signal that behaves like an enable to read/write from that particular bank. The contents read from the BHSR decides the bank to be enabled. Further, the contents of the BHSR can also be used to form a mapping logic between the index of the overall design and the banks index. For example, as it can be seen in Figure 10, when the index of the overall design lies between *512-1023,* bank 2 is enabled and is mapped with index *0-511* of that bank.

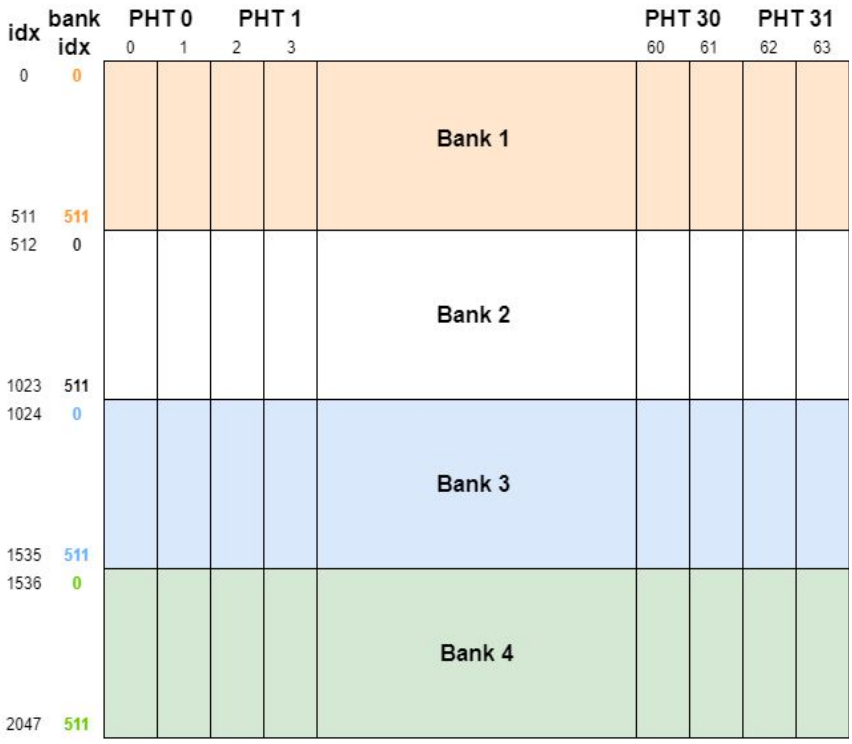


Figure 10: Multi-Banked PHT(SRAM) implementation of a 2-level Predictor with m=0, n=11, k=5

6. Evaluation

Branch predictors offer a very rich space for us to explore - different structures (2-bit FSM vs 1-level predictors vs 2-level predictors), and within those structures, different microarchitectures. There is an excessive amount of design decisions that must be made when implementing a branch predictor: how big must the BHT of a 1-level predictor be in order to obtain maximum accuracy? Do the accuracy returns begin to saturate at a certain size? How do the number of BHSRs, the number of PHTs, and the length of each affect the accuracy of a 2-level predictor? Should one use a register file based implementation or an SRAM based implementation to implement the predictor? We aim to find the answers to these questions (and many others) in this section.

Our plan for dealing with this large design space is illustrated in Figure 11.

We begin with a large design space with lots of possible variations. These variations include the number of bits used to index into the BHT of a 1-level predictor (and hence, the size of the BHT), or in the case of the 2-level predictor - many possible combinations of m, n and k. We will use our Pintool and FL models to sweep through this space and find the accuracies for all of these models. We use these results to pick some of the promising design points and begin

to implement them in RTL. We will consider multiple possible methods of implementing these models - register file based and SRAM based. Next, we push these RTL models through the ASIC flow, which gives us quantitative data for area, performance, energy and cycle time for each of these models. We use this data and combine it with accuracy results in order to motivate our design decisions.

Let's begin at step 1: sweeping through the entire design space to find promising design points.

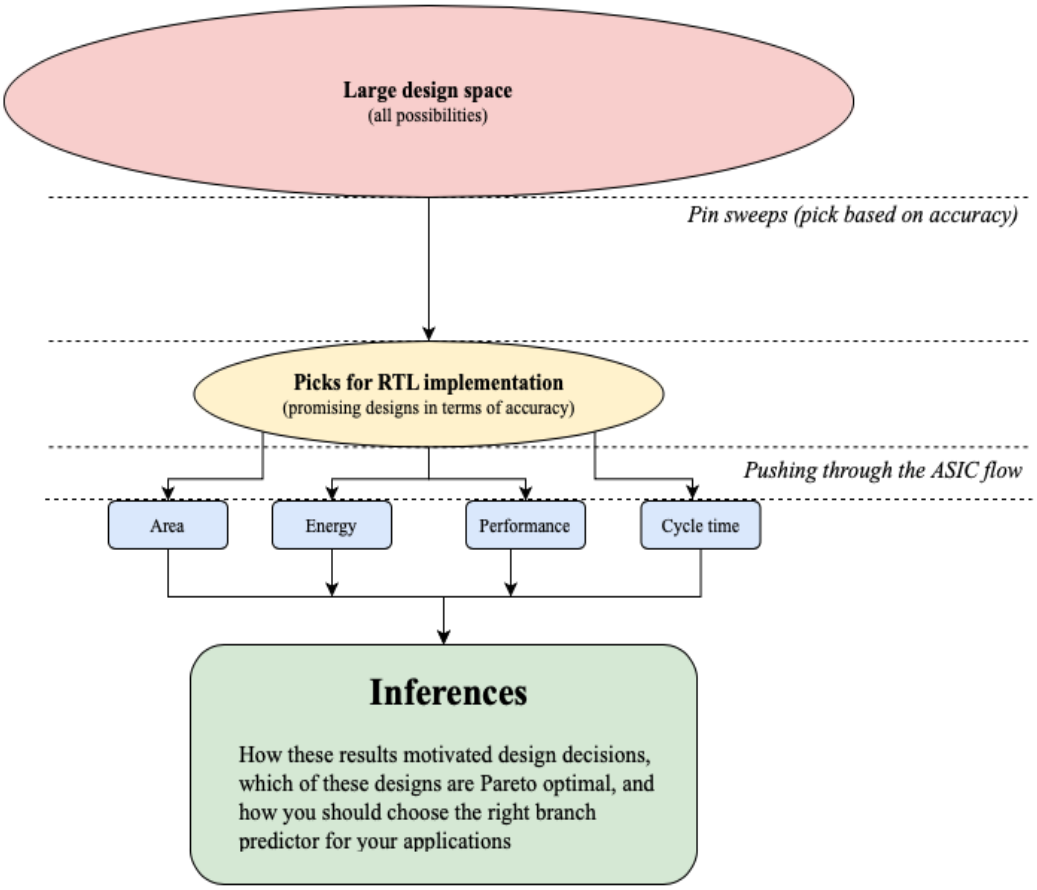


Figure 11: Plan for evaluation: *start with rich space, and focus in on promising designs*

6.1 Accuracy

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			
Memory (bytes)	**Accuracy**		
	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

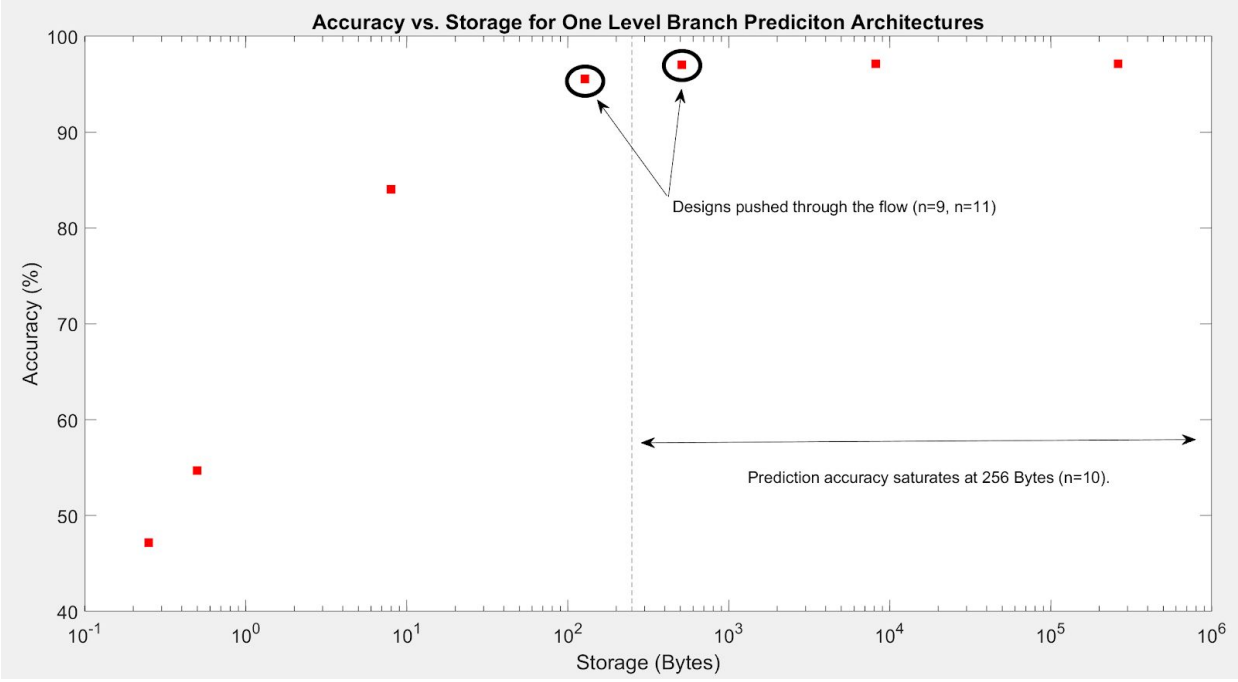
The 2-bit saturating counter has an average accuracy of **60.16%** on our benchmarks. This is a significant improvement over our baseline design, and at the very attractive cost of only 2 bits. It's interesting to note that the 2-bit saturating counter performed better on the smaller benchmark. In order 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.

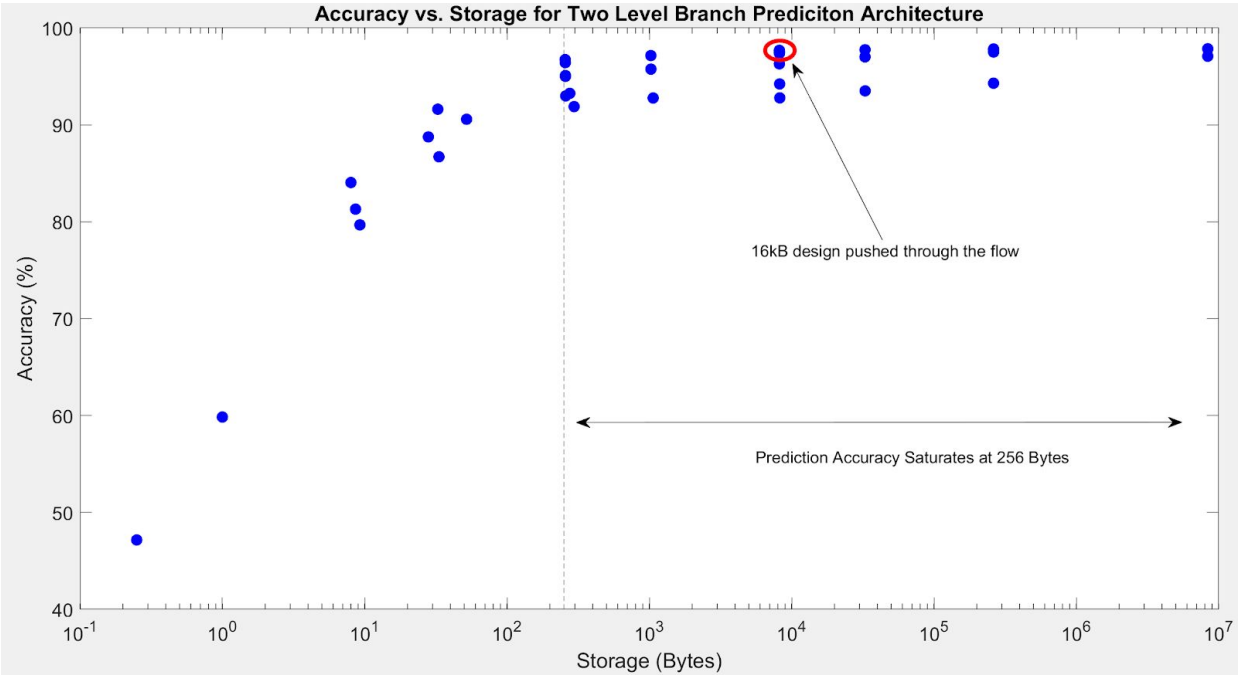


Plot 1: Accuracy vs storage for 1-level predictor variations

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.

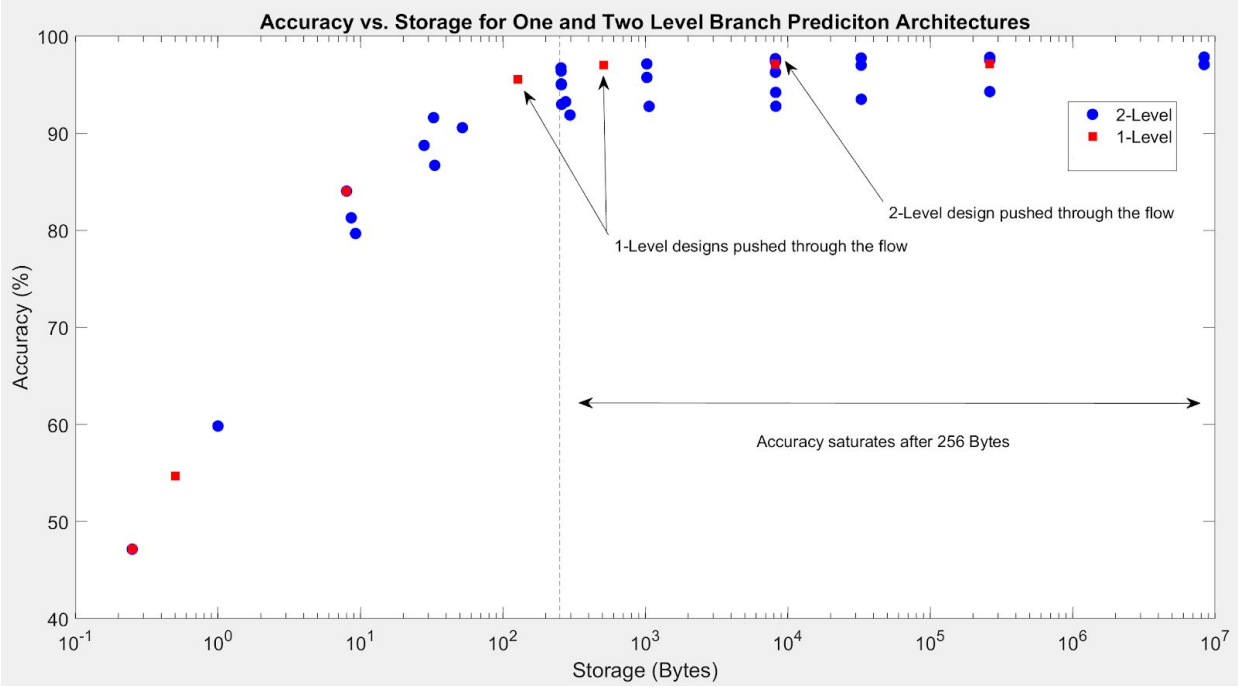


Plot 2: Accuracy vs. Storage for 2-Level predictor configurations

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.

After pushing these designs through the flow, we obtained quantitative results for area, energy, cycle time, and performance. Let's take a closer look at each one in the following subsections.

6.2 Area

In total, we pushed 9 designs through the flow. Out of these, 6 were 1-level predictor implementations, and three were 2-level implementations. The six 1-level predictor implementations that were pushed through the flow were:

1. Regfile based BHT, n = 5
2. Regfile based BHT, n = 9
3. Regfile based BHT, n = 11
4. SRAM based BHT, n = 5
5. SRAM based BHT, n = 9
6. SRAM based BHT, n = 11

We pushed the 1-level predictor with n = 5 through the flow before we had determined that n = 9 and n = 11 were optimal numbers. Nevertheless, the results from the predictor with n = 5 offer some interesting insights.

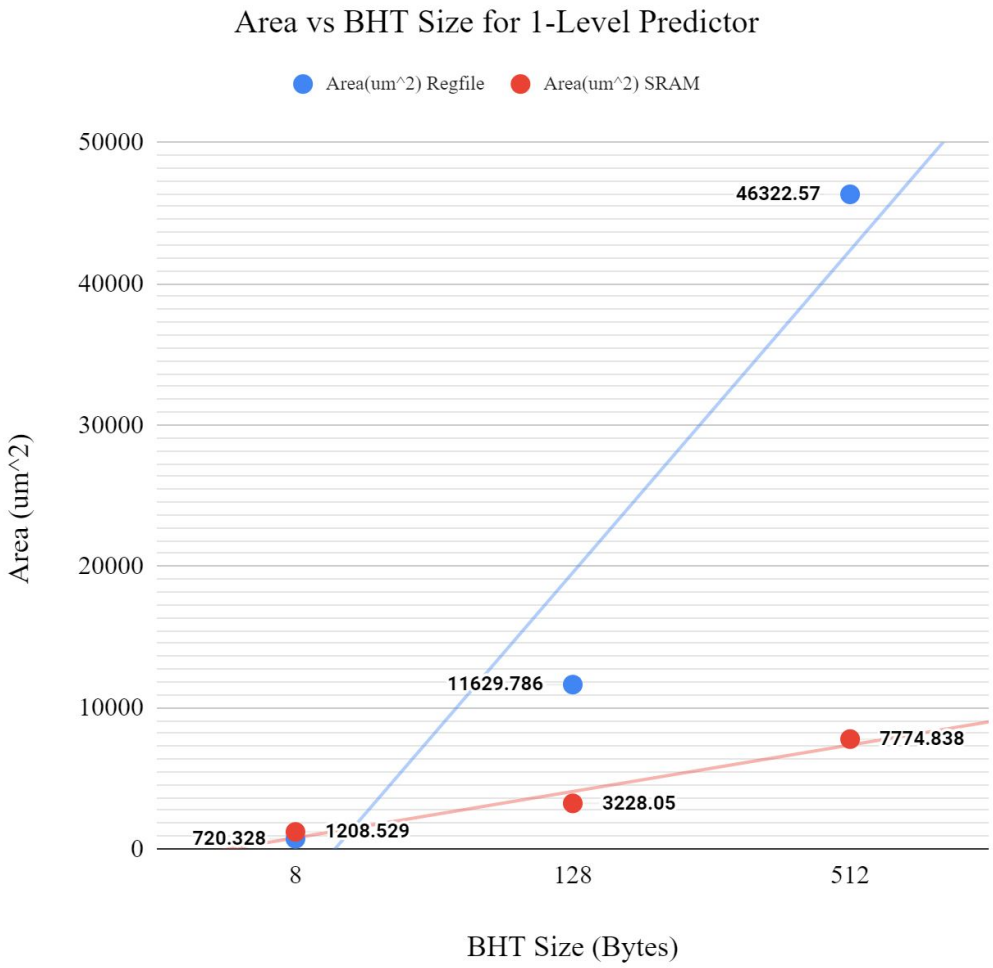
The reason that we chose n = 9 and n = 11, as opposed to n = 10 and n = 12 was because n = 9 and n = 11 are a better choice when trying to implement SRAMs with reasonable aspect ratios. An 11 bit index means 2048 entries, each holding 2 bits, for 4096 bits of memory (64x64 SRAM). A 9 bit index means 512 entries, each holding 2 bits, for 1024 bit of memory (32x32 SRAM).

Let's take a look at the area implications of these 6 designs. Table 4 shows the area results for the 1-level implementations.

Implementation	n	Memory (bytes)	Area(um^2)
REGFILE	5	8	720.328
	9	128	11629.786
	11	512	46322.57
SRAM	5	8	1208.529
	9	128	3228.05
	11	512	7774.838

Table 4: Area results for 1-level implementations

When n = 5, the SRAM implementation has a higher area than the regfile implementation. Conversely, as we increase n, the area of the regfile implementation rises much more quickly than the SRAM implementation's area. In the cases where n = 9 and 11, it is better to go with an SRAM implementation rather than a register file because it reduces the area by ~4 times when n = 9 and ~6 times when n = 11. From Plot 4, the greater slope from the regfile implementation shows that as n increases it becomes more and more beneficial in terms of area to use an SRAM. Only at small values of n, shown from the intersection of the lines of best fit, does the register file consume less area.



Plot 4: Design Area vs n for SRAM and Register File

Figure 12 and Figure 13 show the amoeba plot of the 1-level BHT. The SRAM is depicted by the rectangular grid. It can be clearly seen how even with using the SRAM implementation over the regfile, most of the area is occupied by the SRAM compared to the surrounding logic. The takeaway here is that memory is the major contributor in terms of area, which is why we want to have smaller sized BHTs.

In addition to the discrepancies in area, the regfile vs SRAM implementations also differ heavily in performance, in which the regfile is superior. This will be examined further in the Performance subsection. To summarize, when $n < 6$, the regfile implementation is better not only for area, but also performance. However, for $n > 6$, the regfile area scales very quickly, and the area overhead saved from using SRAMs is worth the performance loss. Because our pintool sweep showed accuracy saturation starting at n=10, there would never be a need to implement a 1-level branch predictor with $n < 6$, and thus we settle on the SRAM implementation for the 1-level design.

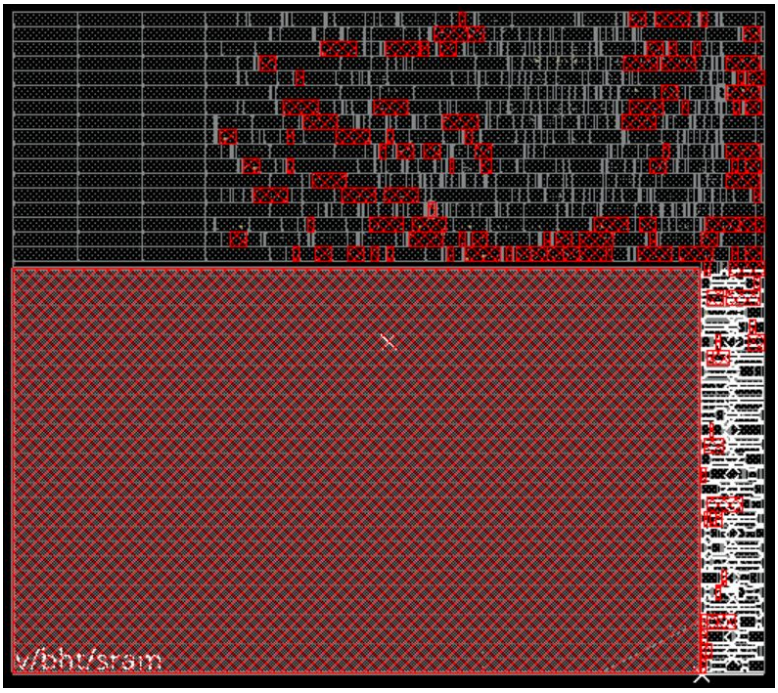


Figure 12: Amoeba plot of 1 level BHT with n = 9 with 32x32 SRAM

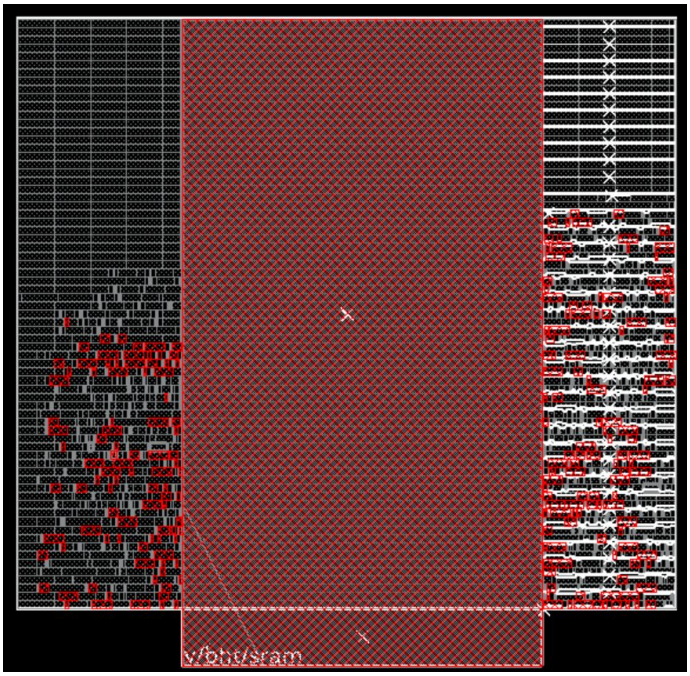


Figure 13: Amoeba plot of 1 level BHT with n = 11 with 64x64 SRAM

We now look at the area implications of the 2-level designs. We pushed three designs through the flow. One design used m = 1, n = 6, k = 5, the second used m = 0, n = 11, k = 6, and the third design used m = 0, n = 11, k = 6 again, but with a multi banked SRAM approach of four 64 x 512 SRAMs. The first design was pushed through the flow to get the 2-level design working, the second design was implemented with the optimal values of m, n, and k found from the pintool sweep, and the third design looked at reducing cycle time by reusing the m, n, and k values from the pintool sweep but using a multi banked SRAM approach. Because the optimal value of m was found to be 0 (capturing global history more effectively), we chose to implement the BHSRT with a regfile, and the table of PHTs with an SRAM. This reasoning stems from the register file vs. SRAM implementation tradeoffs discussed previously. Implementing the BHSRT as a register file allows for single cycle accesses as well as a lower area overhead, and implementing the table of PHTs with an SRAM capitalizes on the area benefits of SRAMs when using a larger memory. The area results for the first two designs are shown in Table 5.

Area for 2-level predictor					
Dimensions			**Memory (bytes)**	**Constraint(ns)**	**Area(um^2)**
m	**n**	**k**			
1	6	5	512	1	7860.224
0	11	5	16384	40	148399.408

Table 5: Area Results for 2-Level SRAM implementation

As shown in Table 5, the area overhead from the 2-level design is very large. A large area is to be expected as our SRAM is quite big. The main issue, though, was cycle time. The large, monolithic SRAM has a cycle time of over 30ns, which is far too slow for our consideration. Because of this, we chose to switch over to a banked methodology. To understand how a banked methodology would affect the area, we considered two SRAM configurations that store 16kB:

1. A single 256 x 512 SRAM
2. Four 64 x 512 SRAMs

We captured information regarding area and CLK->Q delay from the .lib file to get a better understanding of how banking affects the design. Table 6 shows that a multi-banked approach is more expensive in terms of area than using a monolithic SRAM. However, due to the cycle time of over 30ns that the monolithic design gives, we believe this added area overhead to be necessary. We will further examine the effects of cycle time that the multi banked approach had in Section 6.4. Figure 14 shows the amoeba plot from pushing the 2-level design using a single monolithic SRAM, and Figure 15 shows the amoeba plot from using four 64 x 512 SRAMs.

Dimensions			Memory (bytes)	Area(um^2)	Num Banks	Clk -> Q delay
m	n	k				
0	11	5	16384	148399.408	1	32ns
0	11	5	16384	169340.868	4	8ns

Table 6: Area results for banked vs unbanked architecture

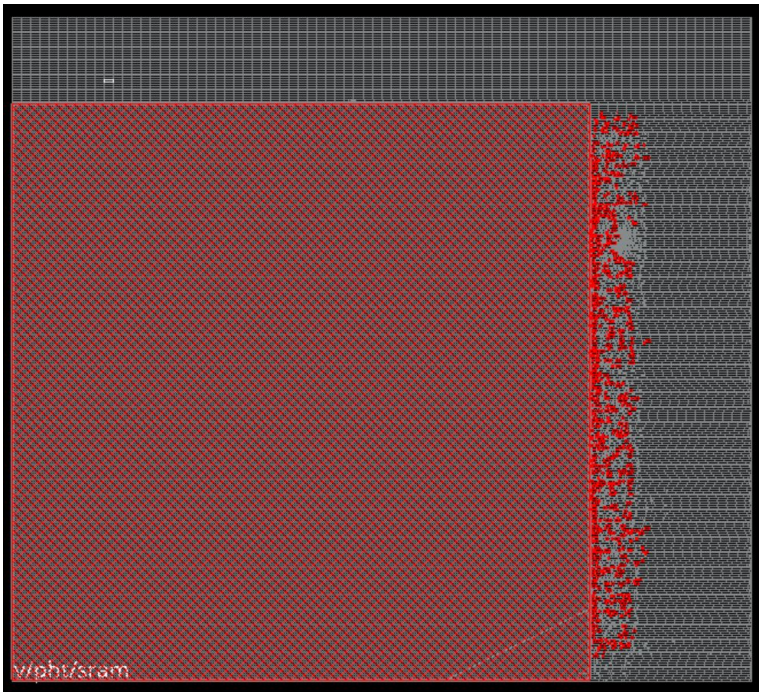


Figure 14: Amoeba Plot of 2 level predictor with m=0, n=11, k=5 with 256x512 SRAM

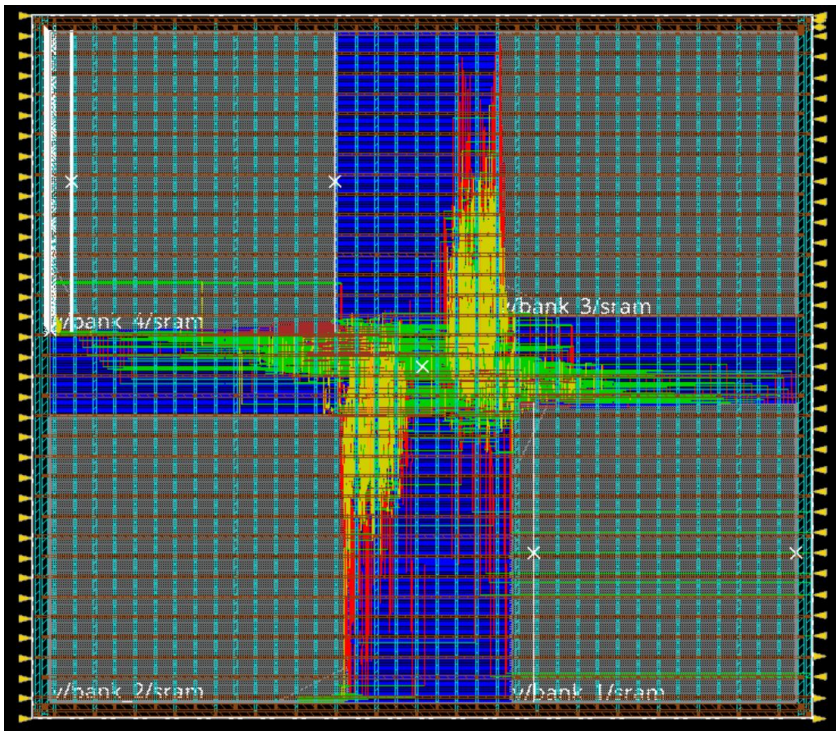


Figure 15: Amoeba Plot of the 2 level banked branch predictor

Note how four banks of equal capacity (64 x 512) is used to configure 64 x 2k predictor

6.3 Energy

The plan for energy analysis to ultimately compare the amount of energy that is required to predict a single branch against the amount of energy that would have been wasted in executing an instruction that would later be squashed in the situation where there was no branch prediction.

In order to do this, we pushed the pipelined processor through the flow and compared how much energy was spent in executing one instruction to how much energy was being spent in one branch prediction-update transaction. Table 7 gives an energy comparison between the processor executing one instruction and the branch predictor performing a predict-update transaction. With these results we can note that the processor spends more energy in executing one instruction than the 1 level and 2 level predictors spend in one predict-update transaction. This is promising, as having a power hungry branch predictor would defeat its purpose.

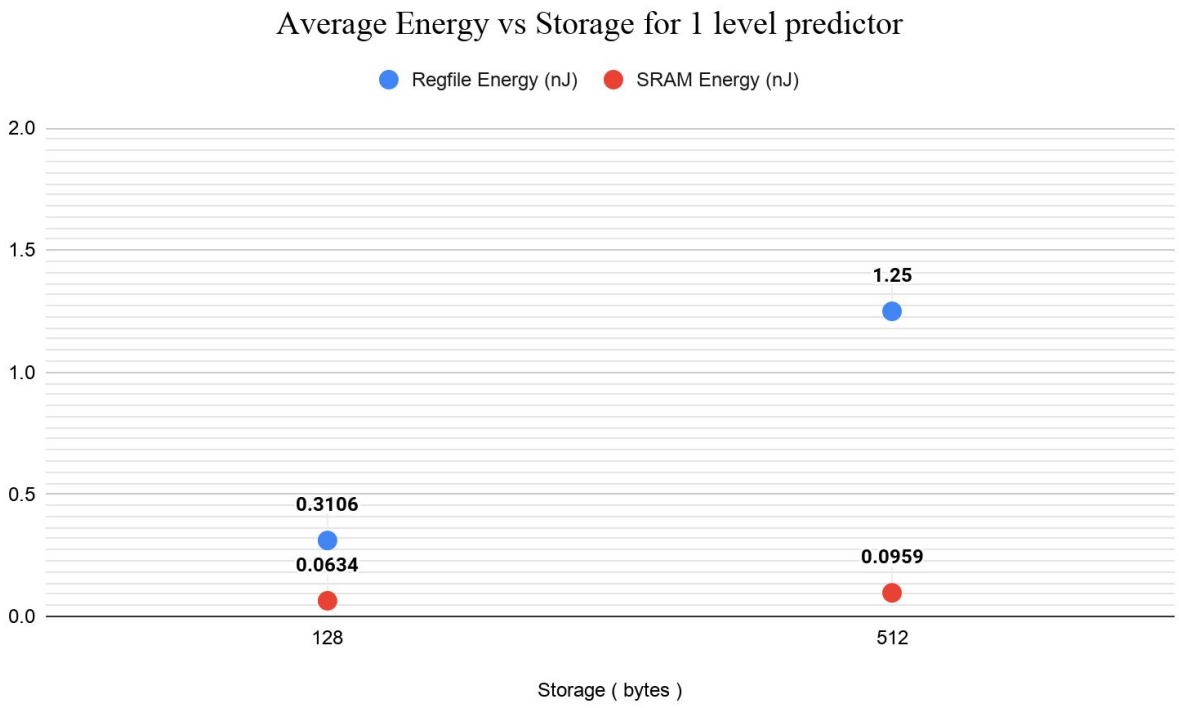
Energy spent per transaction by the processor and predictors					
Metric	**Processor**	**1 Level**	**Relative (1L to Proc)**	**2 Level**	**Relative (2L to Proc)**
Energy/transaction (nJ)	0.014	0.0024	**0.174x**	0.011	**0.785x**

Table 7: Energy/Cycle comparison between Processor and Predictors

Further, we compare the energy spent by the regfile and SRAM implementations of the 1-level predictor in order to understand how size affects the energy consumed. Table 8 tells us how much energy was spent by the 1-level and 2-level implementations in all of our tests. From Table 8 it can be seen how the SRAM consumes less energy than the regfile implementation for all sizes. We get an energy improvement of ~5 times with the SRAM implementation. However, we must also consider the loss in performance due to the increased latency in the case of SRAM. SRAM takes 3 cycles to complete a single transaction over a Register File which takes a single cycle. It is interesting to observe from Plot 5 how energy scales up in the case of a register file as we increase the size of our predictors, whereas for an SRAM based implementation there is very little increase in energy when we move to a bigger predictor. Therefore, it is a good idea to use a register file when the predictor is small, as we do get better performance. But when we move to bigger predictors, the area and energy overhead becomes significant for a register file based implementation as opposed to an SRAM-based implementation.

Implementation	n	Memory (bytes)	Energy(nJ)		
			Single branch in program	Multiple branches in program	Multiple interleaved branches
Regfile	9	128	0.1993	0.2706	0.4619
	11	512	0.8127	1.083	1.856
SRAM	9	128	0.008017	0.08496	0.09739
	11	512	0.03429	0.121	0.1326

Table 8: Energy results for SRAM and Register File for 1-level predictor



Plot 5: Storage vs Average Energy for SRAM and register file implementations

We also analysed the energy consumption vs accuracy trade-off for the 1 level and 2 level predictors. Table 9 gives the Energy and Accuracy comparison between the two predictors. From Table 9, it can be seen that the 2 level predictor consumes ~ 2 times more energy than the one level predictor which is expected because of the choice of the banked architecture where 4 equally sized SRAMs are used. Even though the 2 level predictor is more power hungry, it gives us better accuracy than the 1-level predictor. However, if low energy was a priority, the 1-level predictor would make more sense.

ENERGY COMPARISON BETWEEN 1 LEVEL and 2 LEVEL PREDICTOR		
Metric	**1 Level**	**2 Level**
Average Energy(nJ)	0.0959	0.1605
Average Accuracy(%)	96.5	97.5

Table 9: Energy Comparison between 1 level and 2 level Branch Predictor

6.4 Performance:

Although our branch predictors were implemented in isolation, we must ensure that they do not lie on the critical path when they are incorporated into a processor. In this section, we will take a look at the cycle time and performance implications of the 1-level and 2-level designs, and look at how these metrics motivated some of our design decisions.

Cycle time:

As previously discussed, we can have a register file based 1-level design or an SRAM based 1-level design. The cycle time results for the 1-level implementations are seen in Table 10.

Cycle time for 1-level designs			
n	**Memory (bytes)**	**Regfile constraint (ns)**	**SRAM constraint (ns)**
5	8	0.5	0.5
9	128	0.6	0.5
11	512	0.75	1.1

Table 10: Cycle Time for 1-level designs

The key takeaway here is that both of these designs will remain off the critical path for a processor running at up to 909MHz. This gives us confidence that the designs are fast enough to be incorporated into a processor.

As mentioned in the area section, our 2-level design was a 16kB sized branch predictor. We could implement this design with either a monolithic SRAM or by using 4 smaller SRAMs as banks. We tried both approaches. The cycle time results are shown in Table 11.

Number of Banks	Cycle time (ns)	SRAM Configuration
1	32.508	256x512
4	8.366	64x512
4	8.366	128x256

Table 11: Cycle time for 2-level designs

First, we implemented this predictor using a monolithic 256 x 512 SRAM. This design had a cycle time of 32.508ns, which was far too slow for our purposes. In order to remedy this, we chose to go with a banked methodology that used 4 SRAMs, each of size 64 x 512. This cut our cycle time to 8.366ns. This is a big improvement over the monolithic SRAM implementation but still slow, and will most likely end up on the critical path of most processors.

Performance:

A key theme so far has been the difference in the regfile and SRAM implementations of the 1-level predictor. In addition to area and energy, these implementations also differ in performance. The regfile implementation can perform a predict and update transaction in a single cycle whereas the SRAM implementation takes 3 cycles to finish a predict-update transaction. Using the equation below, we calculate the performance of the SRAM and regfile implementations of the 1-level predictor.

$$Performance = (Cycles/Transaction) \times (Transactions/Program) \times (Time/Cycle)$$

If one transaction is defined as a predict/update operation, the regfile operates at 1 cycle/transaction, and the SRAM implementation needs 3 cycles/transaction. For the same program, the transactions/program will remain the same for both implementations. We also use the cycle time results from Table 10 to determine time/cycle. For a 1-level implementation with a 128 byte BHT, the performance numbers are as follows:

1-level performance (regfile vs SRAM)			
Design	**Cycles/Transaction**	**Time/cycle**	**Performance (ns)**
Regfile-based	1	0.75	0.75
SRAM-based	3	1.1	3.3

Table 12: 1-level performance (Regfile vs SRAM)

The SRAM based design is not as quick as the regfile based design, but as observed in the area and energy sections above, the overall area and energy savings of the SRAM implementation far outweigh this caveat.

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.