# An n-bit Branch Predictor for SimpleScalar

Your name here

**Abstract**—An n-bit branch prediction buffer (BPB) was implemented in the SimpleScalar simulator. Various benchmarks from the SPEC2000 suite were used to test different sized predictors. The results show that predictors any larger than three bits have an either neutral or negative impact on performance.

**Index Terms**—SimpleScalar, Branch Prediction Buffer.

## 1 Introduction

SimpleScalar v2.0 is an open source tool used to simulate real programs to obtain and analyze peformance information of simulated microarchitectures. SimpleScalar was originally developed by Todd Austin at the University of Wisconsin Madison[1]. SimpleScalar implements a number of different branch prediction methods to use when simulating. One of the branch prediction methods implemented is a 2-bit branch prediction buffer (BPB). For our project, we chose to extend that predictor to be an n-bit BPB. We then compared simulations ran for 1, 2 and 3-bit BPBs on 7 seperate SPEC2000 benchmarks to see how the number of bits affected performance. We found that depending on the benchmark, the more bits a BPB has, the more successful predictions it will make. Although, the improvements with each additional bit added was not as large as the preceeding additional bit. This diminished improvement caused us to run another simulation with the 4 and 5-bit BPBs as well. In the end we found that at a certain point for all the benchmarks, the performance would level out and in some cases actually get worse.

## 2 Related Work

### 2.1 Existing Branch Predictors

SimpleScalar currently has five existing branch prediction methods to choose from. They consist of two static and three dynamic predictors. The two static branch predictors are Always Taken and Always Not Taken. As each name suggests, for every branch the branch will always be predicted taken or not taken, depending on the method chosen. The three dynamic predictors are Bimodal, 2-Level, and Combined. The Bimodal predictor is the equivalent to a 2-bit BPB. This is the predictor we chose to edit to create our n-bit BPB. The details will be covered below. Static predictors have no states and thus have the same prediction for all branches. The dynamic predictors have states and thus keep a history of past branches to make the next prediction. Below will cover the three dynamic predictors in more detial.

#### *2.1.1 Correlated Predictor*

The 2-Level predictor is the equivalent to a correlated predictor. A correlated predictor was first invented by T.-Y. Yeh and Yale Patt[2] at the University of Michigan in 1991.

A correlated predictor works by having two elements associated with it. First, each branch has a shift register associated with it n bits long that keeps a history of whether the branch was taken or not taken. There then is a pattern history table that is indexed by a different n-bit value that then stores the prediction for any branch matching that same n-bit history. This branch prediction method is useful for when any repetive pattern exists for branches.

### 2.1.2 Tournament Predictor

The Combined predictor is the equivalent to a tournament predictor. The tournament predictor was first purposed by Scott McFarling[3] in 1993.

A tournament predictor uses more than one prediction method. The Combined predictor in SimpleScalar uses both the Bimodal and 2-Level predictors. It then has a third predicator called a meta predictor that then keeps track of which predictor performs best and uses that predictor to make the current branch prediction.

### 2.1.3 Branch Prediction Buffer

The BPB preditor in its current form was purposed by Jim Smith[4] in 1981.

The BPB consists of a table of a certain size containing entries that branches index to. A BPB table entry will consist of n-bits it uses to make a prediction. Each branch indexes to a table entry using bits from the program counter. The n-bits correspond to how many different states a branch can hold at any given time. The figure below shows the state diagram for a 2-bit BPB (the bimod predictor default in SimpleScalar).

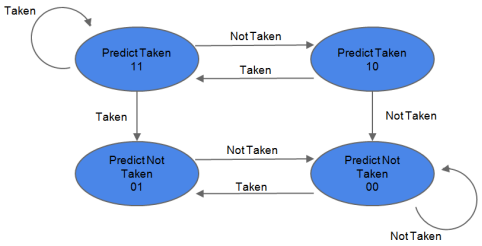


Fig. 1. 2-bit BPB state diagram.

Using the 2-bit BPB as an example. If a branch has the state 10 or 11, the branch will be predicted taken. Alternatively, if the branch has the state 01 or 00, the branch will be predicted not taken. To go from a state 11 (predict taken) to a predict not taken state, the branch will have to mispredict a total of two times. A simple example of why this is useful can be shown by comparing the 2-bit BPB with a 1-bit BPB. The state diagram for the 1-bit BPB is shown below.

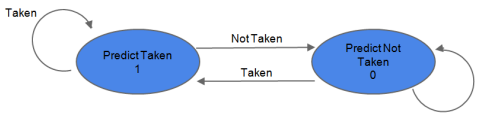


Fig. 2. 1-bit BPB state diagram.

With every misprediction, the next prediction will change from one to the other. An example to consider would be for a loop where greater than 90% of the time, the branch is taken. First considering the 1-bit BPB, when the branch is finally not taken the branch will mispredict and change its prediction to not taken. The next time the loop is encountered it will most likely be taken, but will predict not taken. This would result in two misses each time the loop is not taken. For the 2-bit BPB, in the event the branch is not taken it will mispredict once, but remain in a predict taken state (move from 11 to 10). The next time the loop is encountered it will again most likely be taken and predict correctly. This would result in only 1 miss versus 2 misses, an improvement with the extra bit. The number of states a BPB has is calculated by 2^n. The state diagram for a 3-bit BPB is the following:

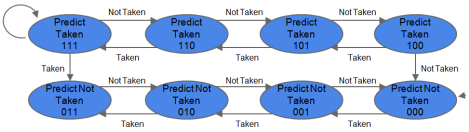


Fig. 3. 3-bit BPB state diagram.

The total number of states it has is 8, making the number of mispredictions needed to change its prediction equal to 4. The higher the value of n is, the greater the number of misprediction needed to change a branches prediction.

## 3 Methodology

### 3.1 Configuration

To implement the n-bit BPB, we added an extra argument to the existing `bpred:bimod` option in the configuration file to specify the size of each table entry.

```
-bpred:bimod <table_size>
-bpred:bimod <table_size> <num_bits>
```

The <num_bits> argument specifies how many bits to use to hold the current state of each predictor in the BTB. To account for the configuration changes, some modifications of sim-outorder.c were made. The size of bimod_config was increased to two to hold our parameter for the number of bits:

```
static int bimod_config[2] =
   { /* bimod tbl size */2048,
     /* nmod bits */2 };
```

Additionally, the call to bpred_create takes this extra element of bimod_config as an additional parameter.

### 3.2 Internal changes

The original bimod predictor used in SimpleScalar was simply hardcoded into the simulator. For example, when checking for predict taken or predict not taken, a comparison was made to see if the predictor's state was >= 2. Or to check if the predictor's state had attained its maximum value the comparison < 4 was used. Our implementation replaces these constant literals with variable values determined by the configured number of bits in the predictor.

Modification was necessary in the following structs and functions declared in bpred.h:

```
bpred_dir_t
bpred_update_t
bpred_create(...)
bpred_dir_create(...)
bpred_lookup(...)
bpred_update(...)
```

The main theme of the modifications is that a BTB entry size variable must be initialized and then used throughout where previously the constant literal values mentioned above were used.

The size variable state_size was added to bpred_dir_t and the size variables dir1_size, dir2_size, and meta_size were added to bpred_update_t. The state_size variable is set in bpred_dir_create which is called from bpred_create. Thus the parameter nmod_bits was added to both these functions. The parameter originates from the new configuration option discussed above and is used as follows:

```
if(nmod_bits)
{
  state_size = 1;
  state_size <<= nmod_bits;
}
else state_size = 4;
pred_dir->state_size = state_size;
```

Thus if it is zero, the default is used and the compatibility with other predictor types is maintained. The remaining modifications follow this pattern of replacing the literal values with the new size variables. A listing of other modifications can be found in Appendix A.

## 4 Experimental Results

### 4.1 Experiemental Setup

Our experimental setup included using SimpleScalar v2.0's out of order simulator. We then tested our n-bit BPB for 1, 2 and 3 bits. We used a shell script that would run each configuration on seven different SPEC2000 benchmarks. The benchmarks used were bzip, crafty, eon, gcc, mcf, mesa, wupwise. We performed two different experiments. In the first we ran and compared the two static predictors with the 1, 2 and 3-bit BPBs. In the second we ran and compared the 1, 2, 3, 4 and 5-bit BPBs. For both experiments we compared performance of instructions per cycle and the misprediction rate.

### 4.2 1, 2 & 3-Bit BPB

For the first experiment we also ran the two static predictors to give a comparison of the performances of the BPBs.

The first thing to notice about the instructions per cycle, Figure **??**, were that apart from the bzip benchmark, the BPBs performed much better than the static predictors. For 3 of the benchmarks the number of bits for each BPB did not make a difference, they were bzip, mcf, and wupwise. The mesa benchmark showed a very slight improvement from the 1 to 2-bit BPB and then a slight decrease from the 2 to

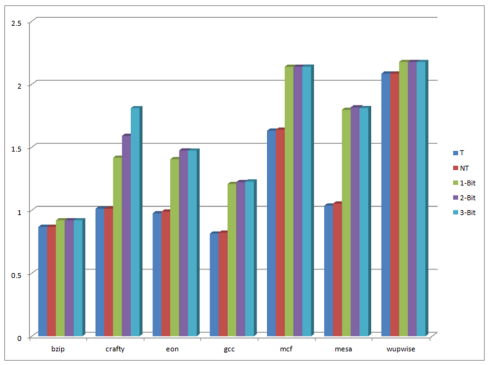


Fig. 4. Instructions per cycle.

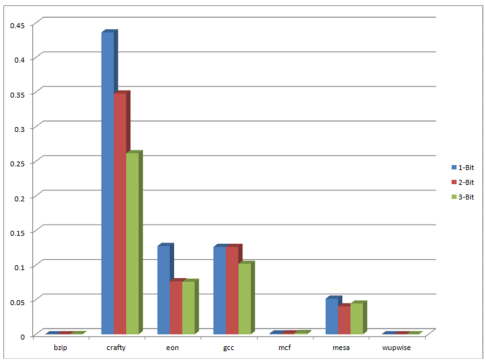


Fig. 6. Misprediction rate excluding taken & not-taken.

2-bit BPB. The eon and gcc benchmarks acted like we expected the results to turn out. They both showed an increase from the 1 to 2-bit BPB and then less of an increase/no increase from the 2 to 3-bit BPB. The crafty benchmark however showed a completely different picture where the increase from the 2 to 3-bit BPB was actually larger than the increase of the 1 to 2-bit BPB.

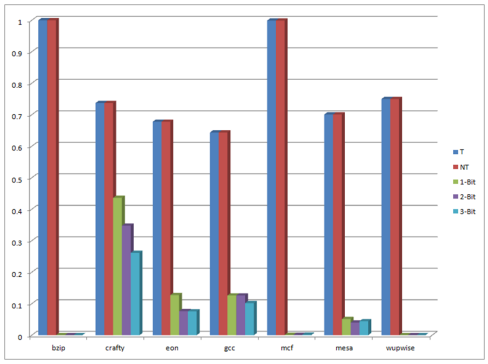


Fig. 5. Misprediction rate.

The misprediction rate, when compared with the static predictors (Figure **??**) show a dramatic difference. The static predictors performed much worse than the dynamic predictors. Figure **??** excludes the static predictors in order to better illustrate the difference among the 1-bit, 2-bit, and 3-bit dynamic predictors.

Among the benchmarks, crafty is the only one to gain a large benefit from when moving from the 2-bit to the 3-bit predictor. Crafty is a chess AI program and likely benefits due to the highly conditional nature inherent in AI programs.

The eon benchmark saw no improvement with the 3-bit predictor and gcc only minor improvement. The mesa benchmark actually performed poorer using the 3-bit.

### 4.3 Additional 4 & 5-Bit BPB

We also decided to run the benchmarks for 4-bit and 5-bit predictors (Figures **??** and **??**). What is interesting here is that benchmarks that improved with the 1, 2, and 3-bit predictors actually degraded with 4 and 5-bit. Our explanation for this is best illustrated with an example:

```
void foo(int flag)
{
  int i;
  for(i=0; i<16; i++) {
    if (flag)
      printf(``Hello'');
  }
}

int main()
{
  foo(1);
  foo(0);

  return 0;
}
```

Consider the state of the entry in the BTB for the line `if (flag)` throughout the execution of the above code. If a 5-bit predictor is used and its initial state is `01111` then at the end of the first call to `foo`, its state will have reached `11111`. Now in the second call to `foo`, that same predictor will mispredict every time. It

will decrement by one at each iteration of the for loop but not return to a state less than `10000`.

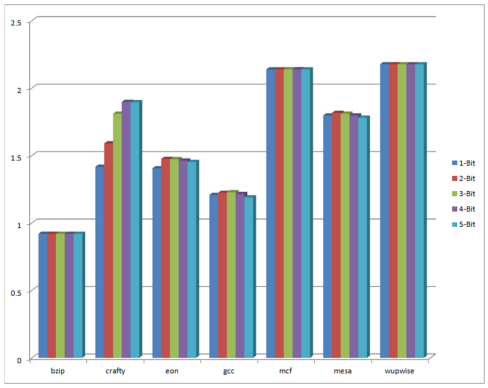


Fig. 7. IPC with 4-bit and 5-bit.

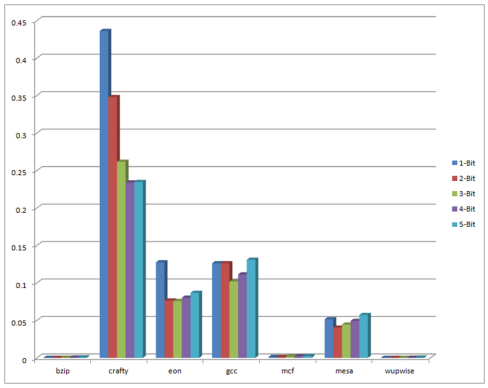


Fig. 8. Misprediction rate with 4-bit and 5-bit.

## 5 Future Work

Future areas of study may be to include the use of the combined predictor with the 3-bit predictor. Also a more detailed cost/benefit analysis of the 2-bit vs 3-bit predictor might be useful as most commercial processors have opted for 2-bit predictors.

# APPENDIX A
# SOURCE MODIFICATIONS

## A.1 sim-outorder.c

1) line 120
2) line 659
3) line 925
4) line 945
5) line 969

## A.2 bpred.h

1) line 121
2) line 180
3) line 182
4) line 184
5) line 192
6) line 215

## A.3 bpred.c

1) line 69
2) line 90
3) line 94
4) line 98
5) line 104
6) line 110
7) lines 200-206
8) lines 600-626
9) line 732
10) line 739
11) line 943
12) line 961
13) line 980


# ACKNOWLEDGMENTS

The authors would like to thank Dr. Xin Fu

# REFERENCES

[1] Doug Burger and Todd M Austin. *The SimpleScalar tool set, version 2.0*, ACM SIGARCH Computer Architecture New 25.3 1997. pp. 13-25.

[2] T.Y. Yeh and Y.N Patt, *Two-Level Adaptive Training Branch Prediction*, Proceedings of the 24th annual international symposium on Microarchitecture. Albuquerque, New Mexico, Puerto Rico: ACM. 1991. pp. 51-61.

[3] McFarling, *Combining Branch Predictors.* June 1993. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf

[4] Jim E. Smith, *A Study of Branch Prediction Strategies.* International Symposium on Computer Architecture. May 1981. pp. 135-148.