Saturating Bit Counter Simulator: Simulating the Effects of Varying Counter Sizes

Syth Ryan
Department of Electrical and Computer Engineering
Iowa State University
syth@iastate.edu

Abstract

There have been numerous methods in using the saturating bit counter algorithm in branch prediction. An attribute that can be modified with the saturating bit counter is the number of bits. I wanted to build a simulator that could aid in deciding what amount of bits is right for projects on a more individual basis.

Expanding on this idea, the saturating bit counter could possibly pair the simulator's best prediction with a compiler. That compiler could be recommended a bit size for their project specific saturating bit counter.

Whenever a prediction is correctly made, we drastically reduce computing time. It is possible to try and run both directions of a branch simultaneously and take the correct side after computation but this is expensive in terms of resources. This tells me that even correctly predicting only a small amount of branch instructions has the potential to benefit the system in computation time.

1. Introduction

In life, we can prepare for many things if we have the prior knowledge. We know that a usual day would entail; waking up, brushing teeth, getting dressed, going to work. Sometimes the usual day could be interrupted, for example there may be a snow storm that prevents you from going to work, and your day has now changed.

Besides the few unexpected events such as a snow storm, history tends to repeat itself. If we have a good knowledge on the past we can better predict the future. If we try to predict what will happen in a day, it would be safer to say that we have a usual day. One snow storm may affect one or maybe two days but shouldn't we still prepare for work tomorrow?

The Saturating Bit Counter tends to follow a similar pattern, the more bits the counter has the more tolerance there is to changing its prediction. This idea has been widely used in branch prediction. One common use for this is the Tournament Predictor where there are two or more types of predictions and based on the Saturating bit counter, it chooses the side more heavily weighted or is correct more than the other.

Sometimes there are programs that require more tolerance than others. We want to find out if modifying the number of bits used for a saturating bit counter for specific programs has any branch prediction improvement.

If the result of manipulating the size of the Saturating Bit Counter significantly increases branch prediction, I hope to move onto adjusting the Simulator to help compilers predict what size of bit to use.

2. Saturating Bit Counter

James E. Smith in his paper *A Study of Branch Prediction Strategies* [1] discussed the saturating bit counter strategy. He described the many prediction algorithms that were currently in place and proposed new strategies of predicting branches that would take a fraction of the space but still have the ability to accurately predict.

The need for branch prediction comes from the increasing need for faster computers. Conditional branch instructions take up valuable time; instructions have to come to a standstill in order to determine what set of instructions to execute next in the pipeline. An idea that has been continuously used to speed up this process is to start execution of a predicted branch path's next instructions. There is a down side to this, incorrect predictions take even longer to correct since there now has to be instructions purged from the pipeline. This is why it is very important to have a high percentage of correct predictions.

There were two strategies related to what is being investigated in Smith's paper [1]. Strategy 6 explained how a one bit history prediction could be used. The simple logic of a single bit equaling zero would me predict not taken and a value of one equaling predict taken. The history bit is determined by the previous branch, if the prior branch was taken then predict taken for the next and vice versa. Strategy 7 is the same basic idea but using a 2 bit and 3 bit counter. *Figure 1* shows a two bit counter. We now have a longer history. The more taken branches in the recent last 2 branches mean we should predict taken. This could solve issues if there are multiple taken branches and only a single or very few not taken branches. There is now more of a tolerance to branch prediction the states of this counter scale up to the nth bit chosen.

Smith explains that the number of bits don't show significant improvement for prediction accuracy larger than two bits [1]. His results are shown in *Figure* 2 for a comparison between a 2 bit counter and a 3 bit counter and *Figure 3* shows the single bit history's accuracy.

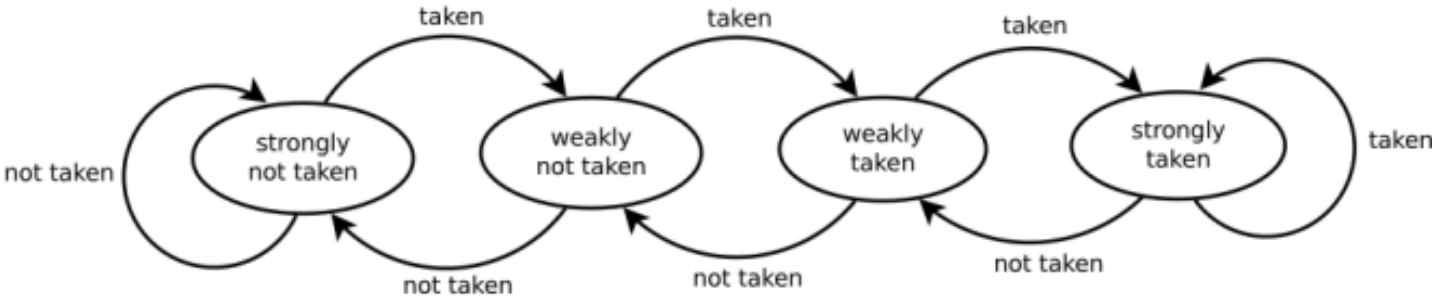

***Figure* 1: an example of a 2 bit saturating counter [2].**

PROGRAM	PREDICTION ACCURACY	
	2 BIT COUNTER	3 BIT COUNTER
ADVAN	99.4	99.4
GIBSON	97.9	97.3
SCI2	98.0	98.0
SINCOS	80.1	83.4
SORTST	84.7	81.7
TBLLNK	95.2	94.6

Figure 2: Comparison of 2 bit and 3 bit counter's accuracy for various programs [1]

PROGRAM	PREDICTION ACCURACY
ADVAN	98.9
GIBSON	97.9
SCI2	96.0
SINCOS	76.2
SORTST	81.7
TBLLNK	91.8

Figure 3 Results of a 1 bit alternating prediction bit for various programs [1]

What caught my eye from these two results were that yes, the majority of the time the 2 bit counter was either equal or slightly better. But the program SINCOS was able to gain a 3.3 percent increase by increasing the counter to 3 bits. This lead to the thought of creating a simulator that could potentially choose a counter bit size based on specific programs.

3. The Simulator

The simulator was compiled using a Microsoft Visual studio 2013 compiler. The code was done in the C programming language. It is currently under preliminary development and can be later refined to aid compilers in choosing the right number of bits for their saturating bit counter.

3.1 The Interface

The current interface for this compiler lets you choose how many bits you would like for your counter. This selection is limited to 1 through 200. This limitation is only due to the preliminary nature of this simulator.

Next on the interface is the size of the table you wish to have which is limited to a range of 1 through 10,000,000. This feature will eventually be taken out as it would become unnecessary for a submitted file of some kind. A file would easily tell us the size from the amount of entries.

Finally a pattern is asked for the user, again as this simulator is in its early stages we have an ad hoc way of entering a pattern. You can enter in a pattern that is repeated up to the table size. For example "tttnntn" is the same as making a table of {Taken, Taken, Taken, Not Taken, Not Taken, Taken, Not Taken}. This is repeated until table size is reached. For the time being this was the best way to do this for a preliminary test.

An example run of the Interface is shown later for the test cases used in Figure 6.

```
7  //#define BASIC_TABLES // uncomment to view basic static table results
8  //#define RANDOM_TABLES // uncomment to view random table results
9  #define CUSTOM_TABLES // uncomment to enable cutom table functionality
```

Figure 4: Simulator configurations

3.2 Running

There are a few options that can be enabled sown in Figure 4 that are currently commented out. These give results of static tables that are easily verified without the simulator. The BASIC_TABLES include an all taken table, a non-taken table and an alternating table. RANDOM_TABLES include two tables populated randomly. The tables we are interested in are the CUSTOM_TABLES which are populated from the pattern entered by the user. A quick example of a custom table being populated would be if a user had entered “ttn” and a table size of 4 the pattern would be {Taken, Taken, Not Taken, Taken}.

The simulator maintains a saturating bit counter state throughout a single table’s execution run. This run is based on a loop that checks whether the prediction (based on the current state of the counter) is correctly matched to the actual, which is acquired from the corresponding entry in the table.

4. Testing

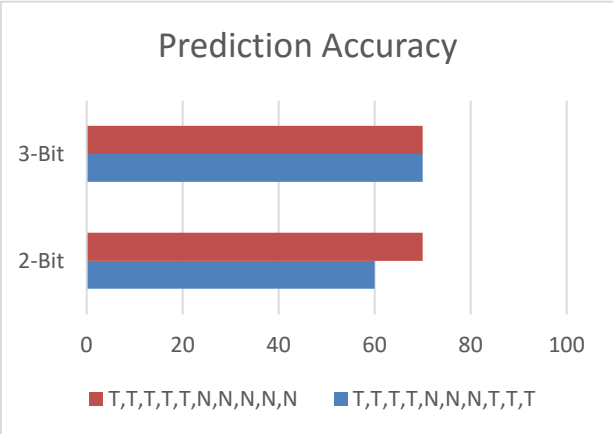

Figure 5: Results of a 2-bit vs 3 bit counter

A few simple tests were ran to show that we can still have improvement on bits higher than 2. The Graph of Figure 5 are results taken from the simulator. The table size was 10 and for simplicity I compared a 2 and 3 bit counter. Here we can see that in the case {T, T, T, T, T, N, N, N, N, N} is predicted with the same accuracy for both counters. But the results of {T, T, T, T, N, N, N, T, T, T} show us that there are still unique cases that the simulator is able to find where a higher bit count than 2 is more affective.

```
----------------------------------------

Choose a size for the Taken / Not Taken tables.
Enter an integer from 1 to : 10000000: 10

Choose a size for the saturating bit counter.
Enter an integer from 1 to : 200: ttttnnnttt

Choose a size for the saturating bit counter.
Enter an integer from 1 to : 200: 2

Choose a pattern for your custom table.
[Example: TTTNNTN
please enter up to 48: ttttnnnttt

Custom:
Hits = 6
Misses = 4

Press enter to start over ...
----------------------------------------

Choose a size for the Taken / Not Taken tables.
Enter an integer from 1 to : 10000000: 10

Choose a size for the saturating bit counter.
Enter an integer from 1 to : 200: 3

Choose a pattern for your custom table.
[Example: TTTNNTN
please enter up to 48: ttttnnnttt

Custom:
Hits = 7
Misses = 3
```

Figure 6: Example execution for the presented test cases in Figure 5.

5. Conclusion

Many Tests conducted in the past show that a saturating bit counter larger than 2 bits doesn’t improve branch prediction accuracy the

majority of the time [1]. There still remains the minority that may benefit from a larger saturating bit counter size. That is why I set out to create a simulator to aid compilers in choosing the right size. The tests show that the simulator is able to detect those minority cases which gives hope in finding out if these implementations benefit from a larger size of bits.

The further I traveled into this simulation, the more I found very similar results to what Smith had displayed [1]. Another detail I hadn't brought into consideration was figuring out the number of bits on a pre run basis without knowing the post run table of branches.

6. Future Work

There is a lot still to be done with this simulator. It currently simulates the prediction outcomes of a pre known branch table. The largest hurdle I see is figuring out how to add an algorithm for using the data the simulator creates, to deciding what size of bits are to be used.

The simulator itself would need some major overhauls in functionality and efficiency. Possibly a GUI would prove more useful for easier use. Also the importing of a set of data from a source, possibly a file.

References

[1] A study of branch prediction strategies In ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture (1981), pp. 135-148 by James E. Smith

[2] "Branch Prediction 2bit Saturating Counter-dia." *File:Branch Prediction 2bit Saturating Counter-dia.svg*. Wiki Media Commons, 31 July 2011. Web. 12 Dec. 2015.

[3] Loh, Gabriel H., Dana S. Henry, and Arvind Krishnamurthy. "Exploiting Bias in the Hysteresis Bit of 2-bit Saturating Counters in Branch Predictors." *Journal of Instruction-Level Parallelism 5* 32nd ser. 5.1 (2003): n. pag. Web. 12 Dec. 2015. <http://homes.cs.washington.edu/~arvind/papers/jilp.pdf>.