Final Project - Advanced Computer Architecture

Álvaro Galisteo Álvarez - e11932434@student.tuwien.ac.at

Introduction

The objective of this project is to implement two branch predictors, the agree preditor and the perceptron predictor and evaluate their performance in addition to comparing them with other predictors already developed. The predictors were simulated and tested in the gem5 simulator running on an Intel 4 core i5 at 1.60GHz.

Implementation details

Agree predictor

The Agree predictor, originally proposed by Eric Sprangle, Robert S. Chappell, Mitch Alsup and Yale N. Pat in *The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference* is a predictor that uses two tables, a pattern history table, and a biasing bit storage, in which each branch has a bias towards a concrete outcome and where each entry in the PHT associated to a pattern determines if this pattern agrees with the result of the bias. This way, the predictor can reduce negative interference (i.e. where two branches have the same pattern but resolve in different outcomes).

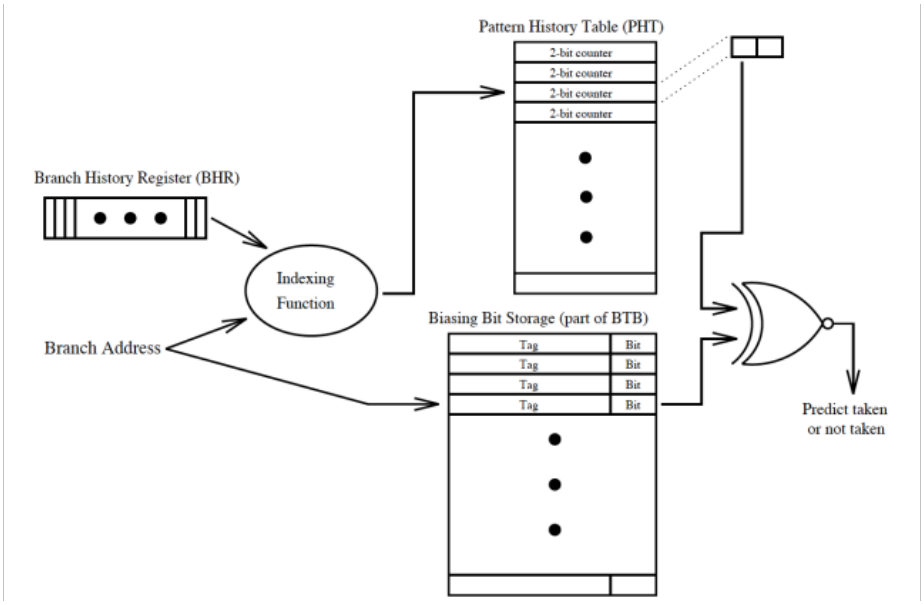

Figure 1: Agree predictor

In our implementation, we speculatively fill a branch history register (BHR) which is used, along the LSBs in the branch address, to index the pattern history table (PHT). The LSBs in the branch address are also used to index the biasing bit storage (BBS). Then, the two results are checked (or XNORed) to see if the PHT agrees with the biasing bit and a prediction will be made. This behavior can be found in the function `lookup(...)`.

On the other hand, the PHT entry is increased if the biasing bit and the outcome were the same (i.e. the bias bit was in the right direction) and decremented otherwise. The branch history register is also restored in case a squash occurred and then updated with the outcome. Finally, the biasing bit entry is also updated to the outcome of the branch, but only the first time the branch is encountered. This behavior can be found in the function `update(...)`.

Moreover, four variables, the size of the PHT and the BBS, the BHR length, and the saturating counter bits are implemented and to prevent an *index out of range* error, a bitmask is used, based on the branch history register length.

Perceptron predictor

The Perceptron predictor, originally proposed by Daniel A. Jiménez and Calvin L. in *Dynamic Branch Prediction with Perceptrons* is a predictor that uses single perceptrons, stored in a table (PT) addressed by the branch address. As this is a binary classifier with a more sophisticated learning mechanism, it provides better accuracy.

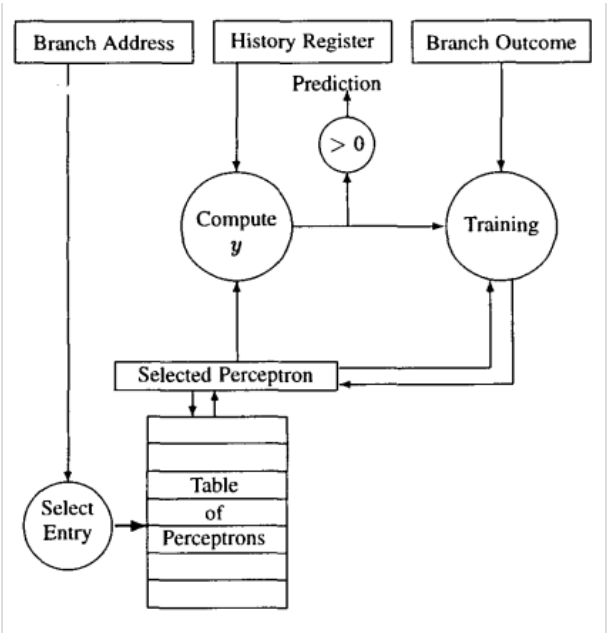

Figure 2: Perceptron predictor

In our implementation, we also speculatively fill a branch history register (BHR) in which k first bits are used as a bias. On the other hand, the branch address is used to index a table of perceptrons. The chosen perceptron is then fed with a biased BHR and outputs a value y. If this value is bigger than *0*, the prediction is *taken* and *not-taken* otherwise. This behavior can be found in the function `lookup(...)`.

On the other hand, the perceptron is trained with a training function (`perceptron.cc`, `Perceptron::train(...)`) when an update is called. In this training, the perceptron is fed again with the biased BHR, along with the y result, the outcome, and a threshold value. This value is extracted from the original paper and its value is calculated as follows:

$$\text{threshold} = \lfloor 1.93 \times \text{globalHistorySize} + 14 \rfloor$$

The branch history register is also restored in case a squash occurred and then updated with the outcome. This behavior can be found in the function `update(...)`.

To implement a perceptron, a new class, `Perceptron`, was implemented. This class contains two public functions `getPrediction(...)` and `train(...)`, along with a private variable where the weights are stored. The training function can be seen in the code, meanwhile, `getPrediction(...)` returns a value based on the sum of the products between weights and the converted bits, where *1* is converted into *1* and *0* into *-1*.

Moreover, three variables, the size of the PT, the BHR length, and the number of bias bits (which must never exceed $\frac{1}{3}$ of the BHR length) are implemented. Also, to prevent an *index out of range* error, a bitmask is used, based on the branch history register length.

Analysis

Note: The PARSEC benchmark suite, as specified in the project presentation was used. The following benchmarks, which all fall in different categories were used: blackscholes, bodytrack, canneal, dedup, ferret, fluidanimate, freqmine, x264.

Agree predictor

Table sizes

Saturating Counter

Perceptron predictor

Table sizes

Biasing Bits

Comparison

First, we must describe the predictors we are going to use for comparison and we must take into account their descriptions when analyzing the results. The predictors used are the following, sorted by their year of appearance:

1. **Local**: The Local branch predictor was introduced in 1991 by T. Y. Yeh and Y. N. Patt. in which the history of a recent branch is used to address a pattern table with saturating counters that predict a branch outcome.
2. **Agree**: This branch predictor was introduced in June 1997 by E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt and is designed to improve a common issue in local branch predictors, where negative interference by different branches occur.
3. **BiMode**: BiMode was introduced in December 1997 by C. C. Lee, C. K. Chen and T. N. Mudge. This predictor divides the prediction tables into two halves, dynamically determines the current "mode" of the program and selects the appropriate half of the table for prediction while preserving global history-based prediction and reducing destructive aliasing.
4. **Tournament**: This predictor is used in the Alpha 21264 processor. Introduced by R. E. Kessler in 1999, this predictor makes use of a hybrid approach, with a local history predictor, a global history predictor and a "chooser" that selects depending on the branch which predictor will give the best prediction.
5. **Perceptron**: Introduced in 2001 by D. A. Jimenez and C. Lin, this branch predictor makes use of single perceptrons, previously defined as binary classifiers to provide better accuracy with a more sophisticated learning mechanism.
6. **L-TAGE**: L-TAGE was presented in a paper by A. Seznec in 2006. It relies on several predictor tables indexed through independent functions of the global branch, path history and or the branch address and usually greatly outperforms the rest of the predictors in sheer performance.

Even though *gem5* has a lot of branch predictors implemented, we choose these predictors are they were the most significant. Every branch predictor was tested in their default configurations and use a similar amount of hardware resources.

From this, we propose the following hypothesis: the most recent predictors will obtain better precision than the old ones.

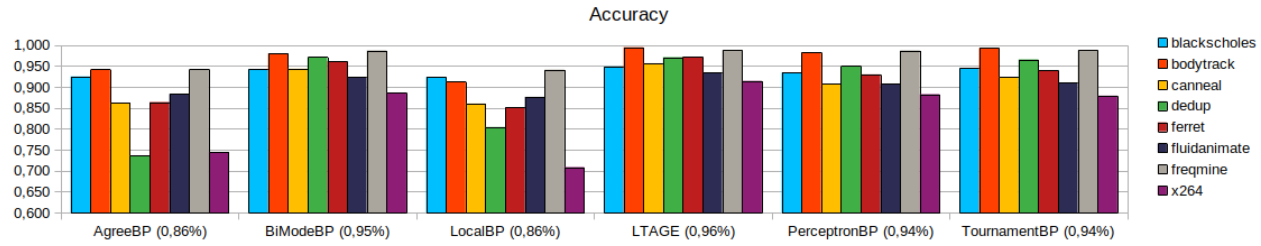

Figure 3: Comparison between branch predictors with averages

In **Figure 3!!!**, we can see that in some of the cases, the new predictors outperform the old ones, as our hypothesis stated, but we cannot confirm it one hundred percent, as on average, the BiMode predictor outperforms the Tournament and Perceptron predictor.

However, we observe that the Perceptron and Tournament predictors have approximately the same accuracy and are near the BiMode and LTAGE predictors, but, our implementation of the Perceptron predictor sometimes falls behind in a couple of benchmarks. Besides, we have observed that being a more complex predictor, the time to perform the lookups and training is longer than others.

Regarding our other implementation, we can see that the Agree predictor improves the local predictor, since, as we have commented before, the Agree predictor tries to avoid negative interferences. However, the improvements don't have a substantial difference.

Another observation we made is that when the branch predictors are more complex, the time they take to make a prediction increases, as more cycles are required to perform lookups and/or training.

It should also be noted that the average accuracy of all predictors is not similar to those shown in their respective papers. This may be due to many reasons, but perhaps the main one is that this researches used different benchmark suites than those used here and that not all programs can provide a certain performance and variations can be very large, however, these tests serve as relatively useful approximations.

TODO: Compare with best results of previous analysis

Conclusion

We can reach some conclusions by taking a look at this analysis. We can say that in general, our implementations require some improvements and optimizations that may not have been conceived during the development of this project, especially in the Perceptron predictor since is a more complex one.

Going into detail, several improvements can be thought of. In the case of the Agree predictor, it has been observed that when the direction of the branch has (a little) more influence when generating the index for the pattern table, the accuracy slightly increases. Thus, we could implement better functions to generate the index to be used in the pattern table, different from a XOR, such as a hash function, a function that gives more influence to the whole direction of the branch or a dynamic function that is training during the execution of the program, such as a perceptron, although we must take into account when making these improvements the number of resources to be used and if some of them are worth sacrificing.