Recreating Results of the TAGE Branch Predictor

CSE240B (WI25) Project Report

Group Details

Name	PID	Contributions
Arjit Verma	A69030364	Review papers, create debug and allocation framework, add tag comparisons, iteratively improve prediction rate
Shaurya Chopra	A69029758	Review papers, create initial working framework, add prediction counters, iteratively improve prediction rate

Introduction

Improving branch prediction accuracy has long been one of the principal ways in improving performance in CPUs. In this work, we analyze in detail the TAGE (TAgged GEometric history length branch prediction) [3] [2] algorithm for branch prediction. TAGE was first proposed by Andre Seznec, and a lot of enhancements have been made on top of TAGE since it was first proposed. These TAGE variants have since become the state of the art for branch prediction in modern CPUs. We reviewed the most recent implementation by Seznec, TAGE-SC-L [1] and have attempted to implement our understanding of the same and assess its performance..

Background and Motivation

Branch prediction is one of the most widely used methods for improving performance in CPUs. Most modern processors are pipelined, and often have pipelines as deep as 20 stages or more. To make full use of the instruction level parallelism that pipelining can provide, it is critical to keep the pipeline full, moving forward with every cycle, and executing the correct sequence of instructions.

Most programs have a fair number of control flow instructions (i.e. branches). A typical number is one in five instructions. This means that, if no optimizations are done, after every five instructions, we need to jump to a different address and start filling up the pipeline fetching sequentially from that address. This would mean instructions currently in the pipeline need to be flushed away, wasting valuable execution time and power. Therefore, if we see a control instruction like a branch, we want to be able to predict which way it will jump at fetch time itself, so that we can minimize pipeline flushes.

As an example, consider the following toy example (adapted from Prof. Onur Mutlu's lecture notes [5]), which is representative of modern CPUs: pipeline depth of 20, and an instruction fetch width of 5 instructions per cycle. Assume that each 5 instruction block ends with a branch. Now, consider different cases of how accurate our BPU (branch predictor unit) is, to compare the effect on the time it takes to fetch 500 instructions:

- 100% accurate
    - It takes 100 cycles, since all instructions are on the correct path
- 99% accurate
    - There are a total of 100 branches in the program. The misprediction rate is 1%, so 20 extra instructions would have been fetched that need to be flushed from the pipeline.

    - This means our program took 120 cycles instead of 100, i.e. 20% performance degradation
- 98% accurate
    - Similar to the above example, now two branches are mispredicted, so 40 extra instructions were fetched and flushed, leading to a 40% degradation
- 95% accurate
    - Five branches are mispredicted, so 100 extra instructions need to be flushed from the pipeline, leading to 100% degradation

The above example demonstrates how critical it is to have a good branch prediction scheme.

In this work, we analyse the TAGE (partially TAgged GEometric history length) branch prediction algorithm. TAGE and its variants are now the state of the art in industry. We analyze and implement TAGE, and compare it with GSHARE, a very famous earlier branch prediction algorithm.

Design

Overview of the TAGE Algorithm

The TAGE algorithm builds upon the significantly simpler GSHARE algorithm, and further optimizes it by using several pattern history tables indexed using a hash of global history registers of varying lengths. The GSHARE (Global History Predictor with Index Sharing) algorithm (which serves as our baseline) uses a hash of the PC (program counter) and a constant length global history vector, h, to index into a table. Each entry in the table is an unsigned 2-bit counter. These are saturating counters, with the most significant bit used for prediction. The hash function is a simple XOR of PC and h. The below diagram illustrates GSHARE [6]:

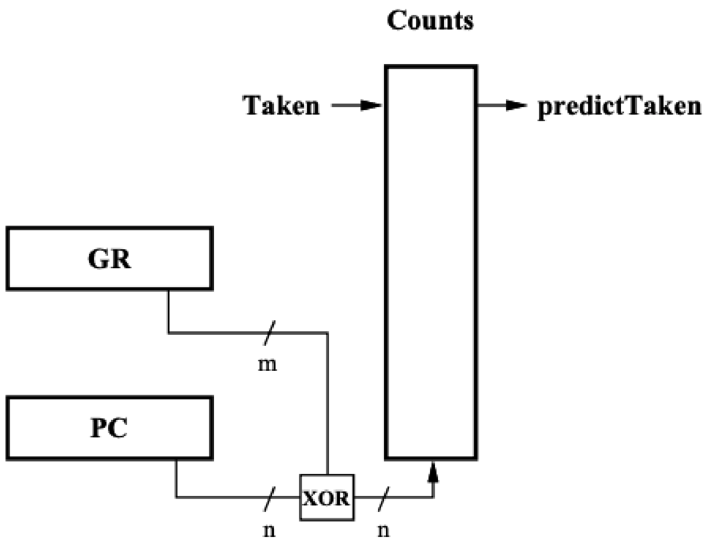

Fig. 1 GSHARE hashing algorithm

In our understanding, one of the key design decisions in GSHARE is the history length itself. There is a correlation between the chosen history length and the performance of the BPU on a particular application. In our understanding, this serves as a key motivation for TAGE. TAGE uses varying, geometrically increasing history lengths, so that it can adapt to different workloads and train the BPU accordingly.

The predictor consists of a “base” predictor, T0, which has a table of 2-bit saturating counters, similar to GSHARE. It then has multiple tables, each of them using geometrically increasing history lengths. If multiple tables return a valid prediction, then the table using the longest history length is used to perform the prediction.

The below figure [3] illustrates a 5-component version:

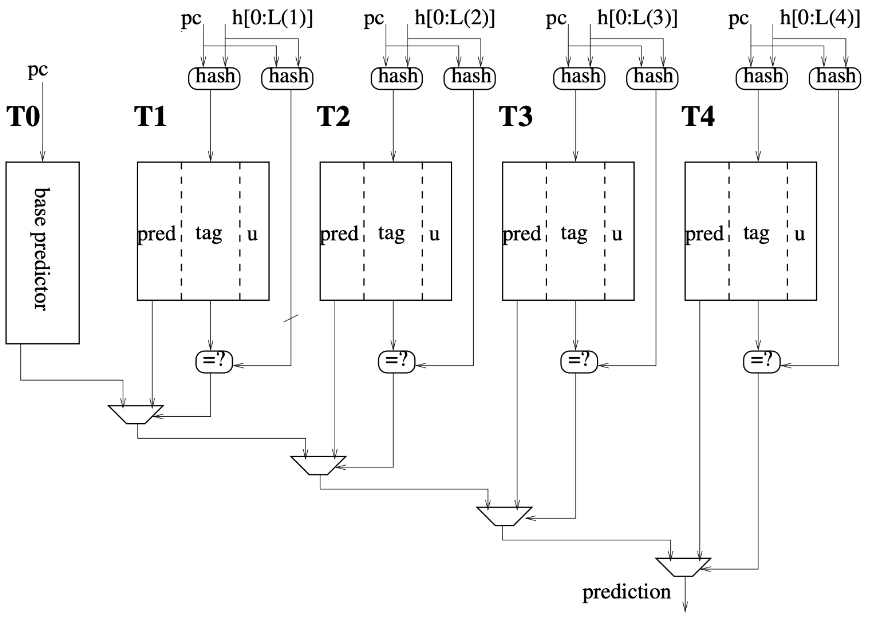

Fig. 2 TAGE branch prediction implementation

Details of Our Implementation

The TAGE SC-L paper [1] implements a 3-bit signed counter for the predictor, a 2-bit counter for the usefulness of the prediction, and an alternate prediction that would come into effect when none of the tables have a corresponding hit for that branch. We noticed that for the instructional traces we were running, there was no significant uptick in accuracy when using a 3-bit signed counter for the predictor instead of a 2-bit signed counter when 2 bits of tag width were used. This may likely be due to patterns saturating earlier in the traces that we were using, and we capitalized on this by using fewer bits for the saturation counter without taking any accuracy hits. Similarly, we have chosen to omit the usefulness counter and the alternate predictor in an attempt to make the implemented design leaner.

We implemented a five component version of TAGE. This includes a base predictor, T0, where each entry is a 2-bit unsigned saturating counter and corresponds to a history length of 10 bits. Our implementation also features four tables of varied history lengths, T1, T2, T3, T4, where each entry has a 3-bit signed saturating counter used for prediction. The counters are all initialized to “weakly not taken” and our chosen history lengths for these tables were 2, 4, 8, and 16 respectively.

In order to fully explore the effect of history lengths, and the BPU adapting to them, we made a small approximation - we did not limit the length of each of the tables to a fixed value. Instead, we allowed them to be as long as required. For instance, if T3 uses 8-bit history lengths, then T3 can hold 2^8 entries. Note that this obviates the need for the eviction policy using the “usefulness” values (although updates to usefulness are still implemented) to simplify things.

We chose the hash function shown in the figure to be an XOR, which is the same as GSHARE. This was ideal for the purpose, as along with being hardware feasible, it also allows the hash function output to retain a notion of both the history pattern and the PC at which the branch occurred. Due to this, the entries in the pattern history tables could be more evenly distributed with reduced aliasing.

We had some discussion amongst ourselves about the “tag” values. The TAGE paper wasn’t very clear on exactly what constitutes the “tag”. We first tried a simple approach of simply using the `hash(pc, h[0:L])` value as the tag. However, when we did not beat the performance of GSHARE, we then discussed, and implemented an improvement of our own on top of this. We instead treated the tables as a “direct mapped” cache of size 2^L, with index calculated as `hash(pc[0:L], h[0:L])` and stored some bits of the PC as the tag instead (refer to the analysis section). We saw performance improvements with this approach, detailed in the results section. Note that this policy still uses a much simpler eviction policy than TAGE - instead of replacing entries by using the “usefulness” index, we ended up shifting to a direct mapped cache approach. However, in our understanding of the original TAGE paper, our expectation was that this should not affect the results, since we were still keeping the tables as big as TAGE would.

Experimental Setup

Our implemented design and the instructional traces are present in the following GitHub repository: *https://github.com/shchopraUCSD/CSE240B_Project/*

We wrote C++ code to simulate the branch prediction architectures that we were attempting to recreate after reading the TAGE papers. While trying to optimize the performance of the implemented branch predictor, we also implemented a simpler Gshare branch predictor in order to compare its performance. To ensure the comparison was fair, we ensured that both predictors used total memory of the same size. In our comparison, both the branch predictors had 2^18 bits of memory. We used instructional traces from several applications such as Deep Sjeng (Chess engine) and x264 (AVC encoder), and this allowed us to compare the performance of the implemented branch predictor in real life applications.

We borrowed the testing infrastructure provided in the following repository to run experiments on the branch predictor we implemented: https://github.com/architagarwal256/UCSD_CSE240A_W25_BP. This infrastructure itself was based on the 2004 Championship Branch Predictor, and unzipped the provided instructional traces and subsequently fed it to the branch predictor model. Kindly note that neither of us were enrolled in CSE240A this quarter and have implemented our own interpretation of the TAGE SC-L paper for the purpose of this project.

Furthermore, since the original TAGE paper [3] mentions improved performance on long traces, we constructed a custom trace, with the original traces first concatenated. Subsequently, to further increase the length of the trace and allow the branch predictor more opportunity to get trained on the traces, we then looped the concatenated version five times.

To better understand the bottlenecks that our implementation was facing, we added debugging counters to keep a track of how many branch instructions were taken vs not taken, the number of branches that were decided by each table, and the number of entries that had been written to each table. This allowed us to better understand if certain tables were underutilized or overutilized, and we could then modify the design accordingly to optimize its performance. We structured our C++ code for optimal readability and divided the sequence into a series of function calls which could then be called to bring the design together.

The simulations can be run on our implemented design with the *--tage* or *--gshare* option from the src/ subdirectory, using the following command:

make && bunzip2 -kc ../traces/long_trace.bz2 | *./predictor --tage*

Results

Global History Bits	Total Storage	Incorrect Branches	Misprediction Rate
17	256Kb	5,262,872	2.631%
18	512Kb	5,758,034	2.879%
19	1Mb	4,845,193	2.423%

Table 1. Results of GSHARE BPU

Tag Bits	Total Storage	T0 Provider	T1 Provider	T2 Provider	T3 Provider	T4 Provider	Misprediction Rate
0	~256Kb	1,215,738	266,606	230,256	769,140	197,518,260	13.151%
2	~324Kb	15,617,213	4,711,358	7,783,618	21,380,618	150,507,193	9.421%
3	~388Kb	62,601,182	1,642,326	3,334,367	9,046,261	123,375,864	7.209%
5	~512Kb	29,893,194	5,139,901	8,581,328	15,613,242	140,772,335	8.399%

Table 2. Results of TAGE BPU

Analysis

We notice in table 1, that GSHARE performs considerably better using the storage that we are currently using, even though TAGE is considerably more complex. Over the several iterations that we ran, we observed very minor changes in the improvement of prediction rate for GSHARE as it is a rather simple algorithm and is more or less directly dependent on the number of storage bits provided. On the other hand, we observed that TAGE showed a lot of promise and kept getting better consistently as seen in table 2. For instance, when we first added the tags, we used higher order bits of the PC. We then experimented with using the least significant bits instead, and found that the prediction accuracy increased. This suggests that the XOR function is actually prone to a fair bit of aliasing.

While we were not able to experiment with as many combinations in the stipulated time to beat GSHARE, the findings of this work are proof that with more debug cycles, the design should be able to beat GSHARE. Our analysis showed that most of the tables T1,T2,T3,T4 were underutilized, suggesting bugs in the allocation policy, i.e. for allocating new entries on a misprediction. We also experimented with different allocation policies, to change the probabilities of choosing tables to allocate new entries in.

We performed one more experiment at the end (results are not part of the table)- adding the usefulness counters' graceful reset feature back. That is, the prediction is used only if it is deemed "useful", and the usefulness counters are reset periodically, with the period set as once in 256K branches. We found that this actually degrades the prediction accuracy from what we saw initially. We also saw in this experiment that the base predictor T0 always ends up being the provider component. One reason for this could be that the BPU is simply not learning the usefulness fast enough, perhaps because the traces simply don't repeat the branches often enough.

Conclusion

Initially, we went with a different understanding of TAGE, and we only implemented the "indexing" logic, as detailed in previous sections. However, on further analysis and discussions, we figured out the approach of implementing both "index" and "tag" logic and this showed significant upticks in the prediction accuracy. As observed in both the above tables, simply increasing the storage budget does not necessitate an improvement in the prediction accuracy. The total storage budget must also be allocated effectively for the BPU performance to improve. This suggests that there exists an optimal balance between the table sizes that could potentially beat GSHARE while using the same storage budget.

Future Work

Future enhancements to our work could potentially include:

- Debugging further as to why the utilization in the various tables T1,T2,T3,T4 is lower than expected.
- Analyzing if, even after the "index" vs. "tag" analysis, our implementation still differs in some fundamental way with TAGE.
- Experimenting with different hash functions - the original TAGE paper was actually not clear about what hash function it uses, and, more importantly, which bits of the PC it uses. We did not have time to experiment with this.
- Changing the base of the geometric series, adding more tables, and trying longer traces to compare the performance with GSHARE
- Adding the Statistical Corrector and Loop predictor mentioned in the TAGE SC-L paper.

While the computer architecture community has widely accepted TAGE and its variants as the state of the art in branch prediction, we still believe that this continues to be an important problem, due to (i) its high impact on performance, as detailed in earlier sections, and (ii) due to ever-evolving workloads and computation demands, which may require tweaks to branch prediction policies, and in the future, even using BPUs in processing elements other than CPUs.

References

1. Seznec, André. (2014). TAGE-SC-L Branch Predictors, Proceedings of the 4th Championship on Branch Prediction, http://www.jilp.org/cbp2014/, 2014.
2. Seznec, André. (2011). A new case for the TAGE branch predictor. 117-127. 10.1145/2155620.2155635.
3. Seznec, André & Michaud, Pierre. (2006). A case for (partially) TAgged GEometric history length branch prediction. Journal of Instruction-level Parallelism - JILP. 8.
4. Undergraduate Digital Design and Computer Architecture Course 18-447 at Carnegie Mellon University, Professor Onur Mutlu, Spring 2015
5. Advanced Branch Prediction at University of Wisconsin-Madison, Prof. Mikko H. Lipasti
6. S. McFarling, "Combining Branch Predictors," *TR, Digital Western Research Laboratory,* Jun. 1993