Performance Comparisons of Low Budget TAGE and Perceptron Based Branch Prediction

Sebastian Thiem
Electrical and Computer Engineering
University of Arizona
Tucson, Arizona
sthiem@email.arizona.edu

Jacob Breckonridge
Electrical and Computer Engineering
University of Arizona
Tucson, Arizona
jbrecken@email.arizona.edu

Andrew Burger
Electrical and Computer Engineering
University of Arizona
Tucson, Arizona
aburger@email.arizona.edu

***Abstract*—This research seeks to compare the performance of the 2KB perceptrons described in [1] and [6] to the state of the art TAGE branch predictor at a similar budget. M. Das found that Jiménez's perceptron based branch predictor [5] outperforms both the L-TAGE [4] and ISL-TAGE [3] at low storage budgets. The TAGE in this paper was implemented and benchmarked on an FPGA, using the design described in Seznec's L-TAGE paper [4] and the Simple-Scalar benchmarking setup in the 1.7KB perceptron paper [1]. This work has found that the TAGE performs at around a 60-67% accuracy at a 2KB storage budget.**

***Index Terms*—Perceptron, TAGE, low storage budget, FPGA**

I. Introduction

A. Background

While the TAGE branch predictor is not found in many processors today, among branch prediction research it is considered state of the art. The journal of instruction level parallelism hosts the Championship Branch Prediction competition, where researchers compete with their branch predictor designs. Across the last 5 competitions, Seznec's TAGE designs have always resulted in top tier performance. It is for this reason that the TAGE is considered state of the art, and why the TAGE was chosen to compare the 1.7KB perceptron to.

B. Motivation

As devices shrink, so too do storage capacities. Embedded devices are limited in the complexity of their branch predictors, due to power and storage limitations [7]. For this reason: the opportunities for a high performing 2KB branch predictor are plentiful in the area of embedded systems.

M. Das's empirical study of branch predictors [2] suggests that the TAGE predictor loses performance at low storage budgets. This is largely due to the exponential storage required for storing the TAGE's tables. Perceptrons, however, have a linearly scaling storage budget that is only dependant on the history length. Due to these differences in how these predictors scale, constraints in the storage budget will heavily favour the perceptron over the TAGE.

II. Related Works

There are 3 key papers that this project is working off of. The first being Moumita Dass empirical study of branch predictors [1], Andre Seznecs description of the TAGE branch predictor [2], and Ali Akoglus perceptron branch predictor [3].

A. The Empirical Study

This study sparked the motivation behind this project. Das examined 5 different predictors at 6 budgets from 64KB-2KB and presented the performance of the predictors at each budget. The study found that while the TAGE performed best at 64KB, it became highly inaccurate around 2KB, in some cases 2x worse than the perceptron predictor at that budget.

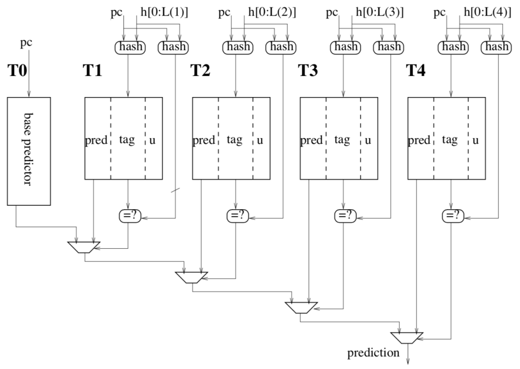

Fig. 1. TAGE Design

B. The TAGE

André Seznec has made many contributions to the Championship Branch Prediction competitions, hosted by the Journal of Instruction Level Parallelism. His most notable predictors were all TAGE based, and for each implementation Seznec has written a paper to support his designs. This is the most descriptive of his papers, and will be used as a reference for the TAGE design in this project. We will not be including the extra predictors in Seznecs designs since the goal of this work is to study strictly TAGE predictors.

C. The Perceptron

This paper describes the perceptron branch predictor that we will ultimately be comparing our results to after this project. At 1.721KB their predictor averaged an accuracy of

93.1%. The 2KB perceptrons described in [1] and [6] boast prediction accuracies of 93.1% and 95% respectively. Using the TAGE module described in Seznec's designs [3][4] and the FPGA based simulation infrastructure described in [1], a direct comparison can be made between the TAGE and the perceptron predictors.

III. Methodology

A. TAGE

The TAGE branch predictor is the result of taking a rather simple idea and adding a lot of functionality on top of it.

With one 3-bit saturating counter, an engineer can make a state machine that will predict branch outcomes based on the behavior of the most recent branches. If the most recent branches are suggesting a trend in one direction or the other the counter will increment or decrement accordingly. This counter works considerably well, however it only predicts based on recent history and will constantly be overwritten with different trends throughout its life. The TAGE seeks to remedy this issue by installing thousands of counters, each with their own association to the current branch history and location in instruction memory. This association is done by hashing.

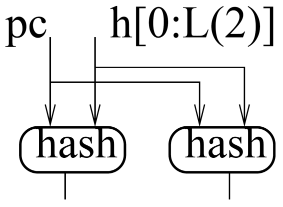

Fig. 2. Hashes

By simply indexing a table with the program counter, one can associate each counter with a point in instruction memory. Indexing by hashing the program counter with the branching history allows each counter to find trends based on both properties. The next step lies within the name TAGE.

$$S = \sum_{i=1}^{tables} \{(ctr + tags[i] + 1) * depths[i]\} + ctr * depths[0]$$

Fig. 3. History Length Formula

The GE stands for GEometric. Certain trends can only be seen by using a short branch history length, while others show up using a long history. The TAGE solves this issue by splitting the counters up into multiple tables, each table with their own history length. The history length for each table is decided by a geometric series and can typically range from 6-2000 bits in large budget predictors.

The TA stands for TAgged. Like other tables indexed by hashing algorithms, the TAGE runs into a common problem: aliasing. Even an ideal hashing algorithm cannot get around the limited table entries. It is for this reason that each counter will have a tag associated with it. The tags will be calculated based on a hashing algorithm different from the indexing hash. This tag hash is also based on the same program counter and branch history as the index, only the output will be calculated differently. By using this second hash value, an entry is considerably less likely to be affected by anything other than the exact association it is watching.

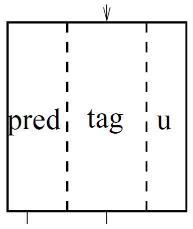

Fig. 4. TAGE Table

At prediction time, an index and tag are calculated for each table. The indices are used to access each tables entry, and the calculated tags are compared to the tags in those entries. Only entries whose tag matched the calculated tag will have their prediction used. If none of the tags match, then a default table will be used. This table has no tags and is just a table of counters indexed by the program counter.

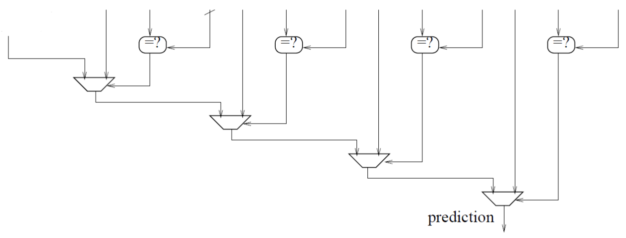

Fig. 5. Tournament Muxes

The prediction from each table is fed into a tournament style cascade of muxes that favors the predictions from longer history length tables over the shorter tables. The largest history length table with a tag match makes the final prediction.

B. Simulation Environment

The 1.7KB perceptron paper [1] measured their performance using the SPEC2000 benchmark suite. In order to generate the traces for the TAGE predictor utilization of the SimpleScalar 3.0 simulator was necessary. The choosing of SimpleScalar over other processor simulators was to keep consistent with the perceptron tested in [1]. The traces can be created by adding a print statement in the bpred_update function of SimpleScalar's branch prediction simulator, sim-bpred. The two variables to print out are the current taken bit and the current branchaddress value. After making this change run the Spec2000 ALPHA ISA benchmarks through sim-bpred.The exact binaries we ran from the spec2000 benchmark suite were from both the integer and floating point benchmark categories. GCC was chosen from the integer benchmarks and EQUAKE/AMMP were chosen from the floating point benchmarks.

After generating those traces, they can be loaded into on-board memory. Then the simulation runs as follows

- Read the next program counter and branch outcome from the trace
- Input the program counter into the TAGE
- Read the TAGE's prediction
- Compare the prediction to the branch outcome
- Update the TAGE based on said branch outcome
- Increment the correct predictions counter if the prediction matches the outcome
- Increment the total branches counter
- Repeat for all elements of the trace

Many of the papers cited in this work use the CBP (Championship Branch Predictor) benchmarks. In these benchmarks the metric analyzed is MPKI (mis-prediction per kilo-instructions). The prediction accuracy in this research was measured as the correct predictions over the total branches; accuracy. This metric provides a less application dependant measure of performance than the MPKI of the CBP benchmarks, and is the metric measured in the 1.7KB perceptron paper [1].

C. Implementation

The implementation of the TAGE was done in verilog, using Vivado to synthesize and run simulations. The core modules of the TAGE were designed based on the descriptions in Seznec's literature [4] and [3]. Those modules being:

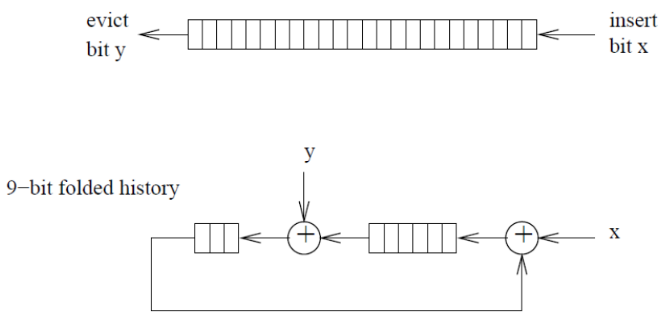

Fig. 6. Hashing Algorithm

1) Hashes: The hashing modules were based on simple xor hashing concepts, the only difference here, is that the two inputs were of different sizes. To handle this, the history is compressed using the design shown in Figure 6. This compression uses the shifting nature of the history to folds the bits on top of each other, allowing a simple method of xor-ing the history and the program counter.

2) Tables: There are two types of tables present in the TAGE design.

The first and base table is a bimodal predictor. This table is indexed by only the program counter and is un-tagged. Just incrementing and decrementing its counters to make predictions. Reads are on the positive edge of the clock pulse, updates are on the negative.

The other tables are tagged, so they contain a tag and useful bit in their entries. These tables are also read on the positive edge of the clock, but their update logic has 4 states controlled by the u_clear, alloc_entry, update_state, and u_set flags. The u_clear will clear all the useful bits in the table. alloc_entry stores the current tag in the hashed entry and sets the u-bit low. The update_state flag increments of decrements the counter based on the branch outcome. Finally the u_set simply sets the u-bit high for the indexed entry.

3) Controller: The controller handles the allocations and updates among the tables. The main logic in the controller just detects the predictor and alternate predictor component. The predictor component is the table that made the prediction this clock cycle. The alternate component is the table that would have made the prediction this round if the predictor component had not predicted.

The provider component has its entry increment/decremented. If the provider component was wrong, then up to 4 entries are allocated in higher history length tables. This is done using a psuedo-random number generator. If the provider component was correct and the alternate component was wrong, the provider component has its entry marked as useful.

4) Tournament Predictor: The tournament predictor has a little more logic added on top of it. Seznec includes a bus that monitors the strength of predictions. He found that in cases where predictions were weak, it is better to take he alternative prediction rather than the provider's prediction [4].

5) TAGE: The TAGE module is a top level module to instantiate the TAGE modules. This design has 6 tagged tables, and 1 bimodal table, the exact specifications can be seen below in table I:

	T0	T1	T2	T3	T4	T5	T6
Depth (log2)	6	7	7	7	8	8	9
Tags	0	7	7	7	7	7	7
History Length	0	6	15	40	102	264	679

TABLE I
TAGE PARAMETERS

IV. EXPERIMENTAL RESULTS

The three traces gcc, equake, and ammp, were run for 100000 branches each. The accuracy measured for those runs can be seen in Figure 7. These accuracies suggest that the TAGE is performing poorly at a 2KB storage budget.

A. Analysis

These results are considerably low, even at this budget. Any results near 50-60% suggest guessing, rather than predicting.

B. Concerns

These results cannot currently be validated under the present infrastructure. Seznec measured his TAGE in MPKI, while this work was measured as miss rates. This was an oversight in this project. Validation is critical for drawing a fair comparison.

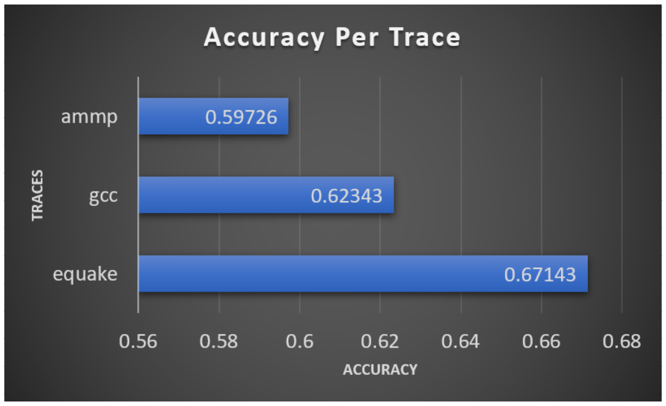

Fig. 7. Benchmarking Results

V. Conclusion

Through the small scale experiment run in this work no conclusion can be made about the performance of the TAGE predictor at a low storage budget.

In order to draw any conclusions validation of the design must be performed. If these performance measurements are valid, then there is a huge argument to be made for the perceptron predictors of [1] [6].

VI. Future Work

Validation is required to finish this work. This validation can be done by obtaining the CBP traces and implementing Seznec's L-TAGE [4]. Once the results are matching the paper, that design can be scaled down to smaller and smaller budgets. Once at 2KB the parameters can be swept to optimize the design, making sure that the design is truly the best representation of an L-TAGE at 2KB.

References

[1] Edward R, Spencer V, Josiah M, John M, A Akoglu. "Balancing the learning ability and memory demand of a perceptron-based dynamically trainable neural network", *The Journal of Supercomputing*. vol 74. pp 32113235. (2018) https://link.springer.com/article/10.1007/s11227-018-2374-x

[2] Das, Moumita, Ansuman Banerjee, and Bhaskar Sardar. "An empirical study on performance of branch predictors with varying storage budgets". *7th International Symposium on Embedded Computing and System Design (ISED)*. pp 1-5. (2017) https://ieeexplore.ieee.org/document/8303913

[3] Seznec, Andr. "A 64 kbytes ISL-TAGE branch predictor". *JWAC-2: Championship Branch Prediction*. (2011) https://www.jilp.org/jwac-2/program/cbp303seznec.pdf

[4] Seznec, André. "A 256 kbits l-tage branch predictor". *Journal of Instruction-Level Parallelism*. (2007) https://www.jilp.org/vol9/v9paper6.pdf

[5] Jiménez et al. "Dynamic branch prediction with perceptrons". *HPCA*, pp197206. (2001) https://www.cs.utexas.edu/~lin/papers/hpca01.pdf

[6] D. Jimenez, C. Lin. "Neural methods for dynamic branch prediction". *ACM Transactions on Computer Systems (TOCS)* 20(4):369397. (2002) https://dl.acm.org/citation.cfm?doid=571637.571639

[7] D. Parikh et al. "Power Issues Related to Branch Prediction" *HPCA '02 Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. (2002) https://www.cs.virginia.edu/~skadron/Papers/bpred_power_hpca02.pdf