Morgan Rae Reschenberg
21 March 2018
CS 152
Laboratory Exercise 3 Undirected Write-Up

Problem 3.2: Branch Predictor Contest: C++ Edition!

We decided to try to build a TAGE (TAgged GEometric) branch predictor. We sourced design information from the RISC-V edition of the course textbook and from André Seznec's paper on the 256Kbit L-TAGE Branch Predictor[1]. We were interested in this design because it seemed to be an expansion on the branch history tables we discussed in lecture with consideration given to global history.

Our branch predictor works as follows: There are 8 tables, 7 of which (tables 1-8) are tagged and one which is untagged (table 0). The untagged table is the default, and is indexed by the PC. The tagged tables are indexed by a hash of the PC and a amount of the history determined by the geometric progression L(i) = 2^i-1, so that the entries in table one are hashed with only the most recent history item, and table 8 is hashed with the whole history. The predictions are implemented as two bits, where 00 is strongly not taken, and 11 is strongly taken. The tables are initialized to weakly not taken.

We use exclusive or for the hash function, but in the case that the amount of history we want to hash is larger than the number of rows in the table, we fold the history down and hash it into the address several times, so that the full history is hashed into the index. This way if two instances of a single branch differ by even one place in the history, they will be mapped to different indexes in the last table and avoid collisions. We let the provider be the table with the longest history where the tags match, and let the alternate be the table with the second longest history where the tags match. We will return the prediction from the provider table, except when the provider prediction is weak, in which case we use the alternate prediction. When we predict correctly, both the provider and the alternate predictors are updated. When we make a misprediction, we allocate at most one new entry in some table. In the case that we miss on table 8, we allocate no new entry and instead just bump the prediction. In the case we miss on table i, where 0 <= i < 8, we allocate space in table i+1 with probability ½, in table i+2 with probability ¼, and in table i+3 with table ¼. The new entry is tagged with the PC and initialized to weakly taken or not taken, depending on the correct direction of the branch.

The journal article we read incorporates several additional features we were not able to include in our simulation. Most notably, the article's predictor has use bits (or age counters) which cause the tables to be periodically cleared. We tried including a similar heuristic in our model, but we were unable to fine-tune it enough for it to become useful. Secondly, the article's predictor is

[1] Seznec, André. "A 256 kbits l-tage branch predictor." *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* 9 (2007): 1-6.

much larger than the one we chose to implement; specifically, the branch history that the article's predictor maintains is several thousands of bits while ours is only 64. To implement a longer history than 64 in C++ would require storying the history in a list rather than in a single long, and the operations on it become unwieldy.

Similar to the article, we found our predictor also hit a wall in terms of performance with respect to tag length. For us, tags below 4b or above 12b worsened performance while tags within that range retained the performance shown. The paper cited similar variance within the 4-8b tag range. We wanted to be able to compare our predictor to the one in the paper and the gshare predictor we were given in the lab, so we chose the following parameters.

Table Length	2^{15}
History Length	15 bits
Tag Length	6 bits
Table Count	8 Tables

The table size and history length are identical to the gshare predictor while the table count is slightly more than the diagramed TAGE predictor from both the book and the article. With these parameters and the given benchmarks, we optimised the tag length both for performance and to fit within the lab's size constraints.

Benchmark	**2-bit MPKI**	**gshare MPKI**	**TAGE MPKI**
gcc	24.429	11.254	5.401
jess	12.759	1.562	0.745
eon	9.631	1.807	1.497
bzip2	0.142	0.094	0.046
Average MPKI	11.59	6.31	4.54

The above table shows the TAGE predictor's performance across four selected benchmarks and compares the performance to the given 2-bit and gshare predictors. It also shows the average MPKI for each predictor. We found that, though the TAGE drastically decreased MPKI when compared to the 2-bit predictor in most cases, it only drastically reduced the MPKI when compared to the gshare predictor on select benchmarks. The gcc benchmark, for example, showed one of the more drastic reductions between gshare and TAGE while the jess, eon, and bzip benchmarks show a large reduction when moving from 2-bit to TAGE, but only a slight reduction from gshare to TAGE.

Overall, the TAGE predictor performed about 7.05 MPKI better than the 2-bit predictor per benchmark and 1.77 MPKI better than the gshare predictor per benchmark.