ECE 209AS Computer Assignment 1 Report

Maxwell Jung (706531056)

***Abstract*—This document outlines the branch predictor used by team "ghtyn" that achieves 3.738 mispredictions per thousand instructions (MPKI) in Computer Assignement 1 for UCLA ECE 209AS course. The branch predictor architecture matches TAGE as outlined in the original TAGE paper but with more complex hashing function that involves both the global history and path history. The C++ implementation of TAGE is imported from RISCY_V_TAGE GitHub and modified to speedup simulation runtime.**

I. Introduction

The objective of Computer Assignment 1 is to simulate a branch predictor in C++. The simulation works by feeding the branch predictor a trace of PC and instructions from a real-world program and counting the number of mispredicted branches per thousand instructions (MPKI). Total of 20 traces are tested and the MPKI results are averaged. The baseline GShare predictor achieves an accuracy of 6.305 MPKI using a history length of 15 and 32k table entries. All simulations were run on an Apple M3 MacBook Pro.

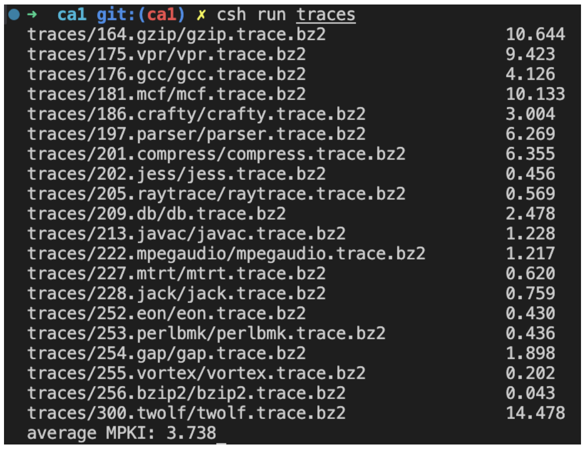
II. Design Methodology

Implementing TAGE in C++ was done by 1) reading the original TAGE paper by André Seznec, Pierre Michaud. and 2) emulating each element of TAGE in C++. Due to the fact that primitive data types in C++ are fixed to 32 bits, custom data structures had to be implemented to emulate branch history and other elements that require an arbitrary number of bits.

I started by implementing TAGE's prediction logic from scatch, however, I realized there exists C++ implementation of TAGE on GitHub. TAGE code from RISCY_V_TAGE GitHub repository by Kanad Pardeshi was chosen because of GPLv2 licensing. Kanad implemented L-TAGE Branch Predictor for a Champsim simulator. I imported the header file RISCY_V_TAGE/branch/tage.h, instantiated it in our simulator, and called the predict() and update() methods to get a working predictor.

I then analyzed every code in Kanad's TAGE implementation to 1) understand how it works and 2) check for consistency with the original TAGE paper. Kanad's TAGE implementation matches 1-to-1 with the original TAGE paper. As a result, I choose to skip the explanation of TAGE as this information can be found in the original TAGE paper and lectures.

The only implementation difference I noticed is the addition of a 4-bit global counter called use_alt_on_na that gets incremented whenever the confidence of the prediction is low (i.e. when the prediction counter corresponds to weakly taken/not taken). When use_alt_on_na reaches a certain threshold (4'b1000 in this case), the alternate component is used instead of the provider component for the final prediction. The explanation for this seems to be that in certain benchmarks, the alternate component provides a better prediction than the provider component.



```
● → ca1 git:(ca1) ✗ csh run traces
  traces/164.gzip/gzip.trace.bz2                      10.644
  traces/175.vpr/vpr.trace.bz2                        9.423
  traces/176.gcc/gcc.trace.bz2                        4.126
  traces/181.mcf/mcf.trace.bz2                        10.133
  traces/186.crafty/crafty.trace.bz2                  3.004
  traces/197.parser/parser.trace.bz2                  6.269
  traces/201.compress/compress.trace.bz2              6.355
  traces/202.jess/jess.trace.bz2                      0.456
  traces/205.raytrace/raytrace.trace.bz2              0.569
  traces/209.db/db.trace.bz2                          2.478
  traces/213.javac/javac.trace.bz2                    1.228
  traces/222.mpegaudio/mpegaudio.trace.bz2            1.217
  traces/227.mtrt/mtrt.trace.bz2                      0.620
  traces/228.jack/jack.trace.bz2                      0.759
  traces/252.eon/eon.trace.bz2                        0.430
  traces/253.perlbmk/perlbmk.trace.bz2                0.436
  traces/254.gap/gap.trace.bz2                        1.898
  traces/255.vortex/vortex.trace.bz2                  0.202
  traces/256.bzip2/bzip2.trace.bz2                    0.043
  traces/300.twolf/twolf.trace.bz2                    14.478
  average MPKI: 3.738
```

Fig. 1. CA1 results

Another feature I noticed in Kanad's TAGE was the relatively complex hashing function used to calculate the index of the component entries and the tags. The hashing function is an XOR of PC, global history hash, and path history hash. The global history hash is generated by continuously folding and XORing the last 12 bits (or 9 bits for tag generation) of history onto itself. The path history hash is generated by performing some integer math on the upper and lower half of path history bits and XORing them together. Replacing these hashes with simpler bit-slicing hashes noticeably drops performance to above 4.2 MPKI.

III. Results

To achieve 3.738 MPKI as seen in Fig. 1, I adjusted the following TAGE parameters: number of components, number of component index bits, number of tag bits, minimum history length, and the geometric series ratio α. After some trial and error, the final design consists of a 16k entries $\times$ 2-bit/entry GShare base predictor and 5 component predictor each consisting of 4k entries $\times$ (3 bit + 9 bit + 2 bit)/entry. Each entry consists of 3 bit prediction counter whose value above 3'b100 corresponds to branch taken, 9 bit tag, and 2 bit usefulness counter that gets reset every 512000 branches. This matches the parameters for 5 component TAGE from the original paper. α is chosen to be the Euler's number 2.71828182846 for no particular reason other than aesthetics and test results. Minimum history length is chosen as 5, matching the original TAGE paper. These parameters form a history length series $\{5, 14, 37, 100, 273\}$.

The total storage budget of this TAGE predictor can be calculated by summing the storage budget of the GShare table

(16k entries $\times$ 2 bit/entry), 5 component tables (5 $\times$ 4k entries $\times$ 14 bit/entry), global history buffer (273 bits), path history buffer (32 bits), and use_alt_on_na counter (4 bit). This totals to 319797 bits $\approx$ 39.04 KiB of storage.

The entire test takes around 30 seconds to run because the global history hash function is a major bottleneck. Kanad's unmodified TAGE code before I modified it performed 7 times worse because the global history was implemented as an array of booleans, requiring each value to be copied and moved whenever a new history value is added. Every calculation of the global history hash also required iterating over the entire array, slowing down simulation. I modified the history data structure such that each array element stores 32 bits of history and can be accessed and moved in one simulation cycle. This allows multiple chunks of history to be shifted or XORed in single cycle, dramatically improving simulation time.