Comparing Branch Predictors

Taylor Foxhall
CS 320

October 30, 2015

1 Purpose

This project is a simple investigation in comparing the effectiveness of different branch prediction methods based on training data. As a baseline we'll look at static predictors and then, perhaps more interestingly, dynamic predictors that react and change according to the training data.

2 Static Predictors

2.1 Always Taken

The algorithm for the static predictors is extremely simple. Essentially we want the predictor to be the program's Yes Man. The entire algorithm is:

```
1 std::string yes_predicitor(const std::vector<Prediction>& preds) {
2     int taken = count_if(preds.begin(), preds.end(),
3             [](Prediction p){return p.taken;});
4     return std::to_string(taken) + "," + std::to_string(preds.size()) + ";";
5 }
```

Code 1: Squirrel!

The function is a simplified calcuation of how many times we take the branch correctly. The acutal logic can be seen in the body of the lambda function we pass to `count_if`, which is a neat utility function from `<algorithm>`. Nothing too complex, just always predict we'll take the branch.

2.2 Always Non-Taken

This algorithm differs trivially from the Always Taken strategy above. Instead we always say no, and the only difference in the code above is that we `return !p.taken;` instead, and we get the exact opposite results.

In general, the results we get from the two methods are highly varied. Since the results of the two are inversely related, if one performs well the other one divebombs, making it a matter or whether or not you guessed more takens than not-takens would occur.

3 Dynamic Predictors

3.1 1-Bit Bimodal Predictor

The 1-Bit Predictor was extremely easy to implement, since indexing the history table was a fairly simple algorithm to implement.

```
// Table for 1-bit predictor
int size = table_sizes[i];
std::vector<bool> table(size, true);
int correct_preds = 0;
for (auto pred = preds.begin(); pred != preds.end(); pred++) {
    int pred_loc = pred->pc % size;
    if (pred->taken == table[pred_loc]) {
        correct_preds++;
    }
    else {
        table[pred_loc] = !table[pred_loc];
    }
}
```

Code 2: Giving our predictor a brain

By keeping track of some kind of history associated with the program counter value, we began to see great improvements in accuracy. As we increased the table size the accuracy made pretty good gains in accuracy, too. However, with each table size increase we also saw diminishing returns, so at some point the benefits will cap off.

3.2 2-Bit Bimodal Predictor

Now we increase the memory to 2 bits for each entry in the table and tells us not to change our minds if we're wrong just once. This changes our algorithm a little, but not greatly.

```
int size = table_sizes[i];
std::vector<unsigned char> table(size, 3);
int correct_preds = 0;
for (auto pred = preds.begin(); pred != preds.end(); pred++) {
    int pred_loc = pred->pc % size;
    if (pred->taken == table[pred_loc] > 1) {
        correct_preds++;
        if (table[pred_loc] == 2) {
            table[pred_loc] = 3;
        }
        else if (table[pred_loc] == 1) {
            table[pred_loc] = 0;
        }
    }
    else if (table[pred_loc] == 0 || table[pred_loc] == 2) {
        table[pred_loc] = 1;
    }
    else if (table[pred_loc] == 1 || table[pred_loc] == 3) {
        table[pred_loc] = 2;
    }
```

```
}
```

Code 3: A few more conditions

The results of this are pretty satisfying, for a small increase in memory we get better accuracy for every table size and the accuracy gains after each increase in size is more consistent and doesn't drop off as fast.

3.3 Gshare Predictor

The idea with the Gshare Predictor was to not only store state assosciated with the program counter, but also remember the result of the last few branches we found to help us more smartly figure out where to update the state of our 2 bits from the 2-Bit Bimodal Predictor.

```
std::vector<unsigned char> table(2048, 3);
unsigned short greg = 0; // Greg will keep track of things for me
int correct_preds = 0;
for (auto pred = preds.begin(); pred != preds.end(); pred++) {
    int pred_loc = (pred->pc ^ greg) % 2048;
    if (pred->taken == table[pred_loc] > 1) {
        correct_preds++;
        if (table[pred_loc] == 2) {
            table[pred_loc] = 3;
        }
        else if (table[pred_loc] == 1) {
            table[pred_loc] = 0;
        }
    }
    else if (table[pred_loc] == 0 || table[pred_loc] == 2) {
        table[pred_loc] = 1;
    }
    else if (table[pred_loc] == 1 || table[pred_loc] == 3) {
        table[pred_loc] = 2;
    }
    greg = (greg << 1 | pred->taken) & r_mask;
}
```

Code 4: Thanks Greg

Oddly enough, the results from this predictor get worse as we have a larger local history. This may be because with a larger history the program is more liable to "confuse" itself. This means that because a smaller global register has fewer possible values it could take, which may translate to more consistent table indexing.

3.4 Tournament Predictor

This predictor is the unholy love child of the 2-Bit Bimodal predictor and the Gshare Predictor. Essentially it's trying to balance the losses from "confusing" the history table from Gshare's methods, but also compensating for the diminishing returns from a large table history in the 2-Bit Bimodal model. It keeps track of which method to use via another history table like in the 2-Bit model. The code is not shockingly different than what's already been shown, so it's not been recopied here.

This predictor consistently beats the best results from both 2-Bit Bimodal and Gshare. This seems like the most accurate version, but I have trouble believing that managing 2 history tables and updating their state in a CPU doesn't inflict some kind of significant overhead, either in complexity or speed.