Spenser Haddad

ECE 513 Computer Architecture Lab 2 Answers

1. The if-else... tree used to update the statistics after every branch in the `updatePrediction` function are not ideally arranged. With some work, they can be moved to remove one of the condition checks. This one fix does have a small effect on performance, but not enough to warrant the readability loss.

2. My implementation is based off the methodology described in Scott McFarling's "Combining Branch Predictors" and Yeh and Patt's "Alternative Implementation of Two-Level Adaptive Branch Prediction". The main branch predictor contains two sub-predictors, which the main one delegates prediction to based off its recent performance. The heuristic for determining which one to use is based off the two-bit saturating counter seen in class. If the counter is below a certain threshold, it uses the first predictor. Otherwise, it uses the second. The base predictor updates the counter after each prediction in its `updatePrediction` method. For each branch, the predictor uses the following system to choose which sub-predictor to use, and how to update the counter after the prediction:

Two-bit Counter Value	Predictor to Choose	On Success	On Failure
0 (00)	Predictor 1	Do nothing	Increment by 1
1 (01)	Predictor 1	Decrement by 1	Increment by 1
2 (10)	Predictor 2	Increment by 1	Decrement by 1
3 (11)	Predictor 2	Do Nothing	Decrement by 1

The first predictor used is an implementation Yeh and Patt's two-level adaptive predictor, specifically their "Per-address Branch History Table and a Global Pattern History Table", or PAg, implementation. As the name implies, there are two levels to the predictor. The first is an array of shift registers that detail each branch's direction history. When a branch instruction is encountered, the lower bits of its address are used as an index into the array. The history tables are *n*-bit integers that represent which direction the branch took on its last *n* executions, where 0 means the branch was not taken, and 1 means it was. For example, the history table "00001111" means that the branch was taken the last four times, but not the four before that. This history table is then used as another index into the second layer: an array of two-bit saturating counters. Unlike the history tables, which are for the most part specific to a particular branch instruction (although instructions with identical least significant bits will overwrite each other), these counters are global to all branch instructions. The predictor then uses this counter to make the actual prediction. After the prediction, the corresponding counter is updated with respect to the actual result as a typical two-bit counter, and the history table is bit shifted left by one, and the least significant bit is then set to the result of the most recent branch.

It's worth noting that there is another theoretically more performant variant of the two-level predictor, called the "Per-address Branch History Table and Per-address Pattern History Tables ", or PAp. Unlike PAg, PAp has multiple tables of counters for each branch address, just like how each branch has its own history table. This gives increased resolution of the behavior of the branch, but also requires significantly more memory than almost any other implementation. With the memory limit imposed, PAp was not found to have significantly better performance than PAg, while using more memory than was reasonable.

The second predictor is McFarling's gshare predictor. Similar to the two-level adaptive predictor, gshare uses a history table shift register to index into an array of saturating counters to make a prediction. However, where the two-level predictor has an array of history tables mapped to different branches, gshare has one global history table. To find the index in the counter array for a branch, the predictor makes a simple hash by XORing the history table with the address of the branch. This allows each distinct branch and history combination to have its own index, assuming the counter array is large enough to fit all of them.

The idea behind combining predictors is that different predictors work better in some scenarios than others, so putting them together allows them to cover each other's weaknesses. Gshare and PAg work well together for this reason. PAg is good at tracking each branch's local behavior, so it can individually adapt each one. However, this means it has less knowledge about how different branches interact with one another, so larger program-flow changes could cause several mispredictions before the predictor adapts. This is where gshare helps. By only having one history table, gshare is good at keeping track of global branch history, so it can identify these changes and adapt faster than PAg. It also helps that, according to McFarling, this combination of predictors has some of the best performance for larger memory sizes, which the 33 Kb memory limit qualifies as in this context.

Finally, the memory allocated to the two predictors was divided up to give each the best performance possible under the constraints. PAg, unfortunately, requires considerable amounts of memory to keep track of all the branches. It's two tables take up memory quickly, and if not carefully watched can easily exceed the maximum allowed space. After some empirical testing, it was found that 512 elements for each of the arrays was the minimum threshold needed to get acceptable performance. Performance could be improved if more memory was available, but it gave diminishing returns that were exceeded by combing with gshare. Gshare needs relatively little memory to work reliably, with most of its space going to its table of counters. The table can't be too small, however, or the address/history indices will collide and overwrite each other. Past this point, however, more memory does little to improve its performance, as many of the indices will never be used. After some empirical testing, 2048 elements seemed to be enough to keep duplicate indices to a minimum, while also fitting into the space left over by the PAg predictor. In total, the memory came out to roughly 25 Kb, with the majority of the memory coming from the two 512 element arrays and the one 2048 element array, all of 8-bit unsigned integers. This leaves enough headroom that any miscalculation won't cause the result measured in the competition to exceed 33 Kb.

3. Given enough cycles and memory, this predictor could certainly be implemented in hardware. Practically speaking, however, it probably is not feasible. Each prediction requires, at minimum, a comparison, some integer arithmetic to calculate indices, and either one or two table accesses, depending on which predictor is used. Updating the predictor requires even more comparisons to update the selection state machine. The hardware required to do all of these operations within one cycle, while not impossible, would be expensive for a single component. In particular, reading from multiple tables would take precious time that could slow the entire pipeline down. Ultimately, it depends on how important the predictor would be to the architecture.

4. Most predictors would not be able to accurately guess the first few passes for any new branch, since they rely on its nonexistant past behavior. The PAg predictor, in particular, requires several branches to initialize its history tables. As a result, it may be easier to "guess" while you let the predictor's reach steady state. In addition, it may not be feasible on certain architectures to always fetch a new prediction, or to send the results back to update the predictor without slowing down the rest of the pipeline. In these cases, the predictor may not update immediately, and it may have to make several predictions without being able to update its state. Considering how dependent each of the combined predictors is on the past history of the program, not receiving updates after every branch would decrease their performance. In this case, it may be wise to replace one of the predictors with a simpler, less history-dependent one, such as a two-bit saturating counter, that can update more frequently and are more robust to receiving lack of changes. Of course, if the predictor is high enough priority, the pipeline could prioritize a loopback to the predictor more frequently.