6.823: Lab 3 Questions

Nada Amin
namin@mit.edu

Due: 7 November 2008

1. When possible, it's more efficient to move code out of the analysis function and into the instrument function, because the instrument code is executed once, while the analysis code is executed repeatedly.

   Since we already know whether a branch has been taken or not during the analysis, it is more judicious to have two analysis functions, one for actually taken branches and one for actually not taken branches, so that when we collect the statistics, we can avoid an extra if statement.

   I haven't noticed a big speed-up from this change. Perhaps, the overhead of an extra if statement per analysis is rather negligible compared to the overall overhead per analysis.

2. My implementation is based on [1].

   I keep a table of N perceptrons. I also keep track of h bits of global history, which records whether each of the h most recent branches are taken or not taken. A perceptron learns to compute $t(x_1, \ldots, x_h)$: given the h bits of global history, it decides whether the target branch is taken or not. The perceptron has $h + 1$ weights, one for each history bit and one for the branch bias which is independent of the history. If it fails to predict the outcome correctly, the perceptron updates its weights.

   The perceptron computes whether the branch is taken as follows:

   $$w_0 + \sum_{i=1}^{h} w_i \cdot x_i > 0$$

   where each x_i is 1 or -1 depending on whether the ith most recent history bit indicates taken or not taken.

   The perceptron updates its weights when it fails to predict the actual outcome. Let t be 1 if the branch is actually taken and -1 if the branch is actually not taken. Then, the weights are updated as follows: $w_0' \leftarrow w_0 + t$ and $w_i' \leftarrow w_i + t x_i$ for $i \in \{1, \ldots, h\}$.

   I chose this perceptron-based algorithm for several reasons:

   - The algorithm is elegant and straightforward to implement.

- It combines global history and local history judiciously. I only need to explicitly track the global history, and the local characteristics are encoded in the weights. Thus, this algorithm is rather efficient in memory usage.
- It outperformed all other algorithms which I tested with similar memory usage.

I implemented other algorithms, inspired by [2] and the class discussion: a bimodal branch predictor, a local branch predictor, a combined local and percetron branch predictor. The perceptron predictor outperformed the other algorithms because of its efficient use of memory and its judicious combination of local and global history.

I decided to use INT8 for each weight. Each perceptron is simply an array of $h+1$ weights. I keep a table of N perceptrons, represented as an array of N arrays of $h+1$ INT8. I also keep a UINT64 (64 bits) for the global history. Thus, the global memory usage is

$$8 \cdot N \cdot (h+1) + 64$$

I tried the following combinations of N and h, each totalling $32832 < 33792 = 33K$ bits of global memory:

- $N = 2^6, h = 63$
- $N = 2^7, h = 31$
- $N = 2^8, h = 15$

Each of these combinations gave me an average accuracy in the 95% range for the benchmarks. The middle combination performed best, so it's my final choice.

3. As explained in [1], it is perfectly possible to implement the perceptron predictor in hardware. In addition, several optimizations are possible: for example, since the inputs x_i are just 1 and -1, it is not necessary to perform full multiplications when calculating the outcome.

   If implemented in silicon, the challenge is to keep the prediction calculation fast enough to complete in a cycle. If that's not always possible, we can use the percepton predictor in conjunction with a simpler faster predictor. If the percepton predictor contradicts the simpler faster predictor, it overrides it, thus potentially catching the mispeculation earlier.

4. In a real machine, several predictions might be made before an update occurs because of pipelining. Indeed, a second branch might follow a first branch so closely that it is necessary to speculate on the second before the first has been (dis)confirmed. Similarly, a third branch might follow and so on. In particular, such a scenario could occur if a very tight loop is

consistently speculated so as to loop, creating a serie of speculations from the same branch.

The impact on prediction is that the history needs to be updated to take into account speculation so that predictions can be made consistently regardless of whether previous branch outcomes are merely speculated or confirmed. For my predictor in particular, I would need to update the global history speculatively with the predicted outcomes. Then, in case of mispeculation, I would need to undo the mispeculated outcome (and all following outcomes) in the global history. This recovery mechanism would require extra memory to be able to shift back the older history bits. The number of extra history bits needed for recovery is bounded by the maximum number of predictions that can occur before the first is updated.

References

[1] Daniel A. Jimenez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20:369–397, 2002.

[2] Scott McFarling. Combining branch predictors. *WRL Technical Note TN-36*, June 1993.