A PPM-like, tag-based predictor

Pierre Michaud

Abstract

The predictor proposed is a tag-based, global-history predictor derived from PPM. It features five tables. Each table is indexed with a different history length. The prediction for a branch is given by the up-down saturating counter associated with the longest matching history. For this kind of predictor to work well, the update must be done carefully. We propose a new update method that improves the mispredict rate. We also propose a method for implementing hashing functions in hardware.

1 Overview

The predictor proposed is a global-history predictor derived from the PPM. PPM was originally introduced for text compression [2], and it was used in [1] for branch prediction. Figure 1 shows a synopsis of the proposed predictor, which features 5 tables. It can be viewed as a 4^{th} order approximation to PPM [6], while YAGS [3] can be viewed as a 1^{st} order approximation.

The leftmost table on Figure 1 is a bimodal predictor [4]. We refer to this table as table 0. It has 4k entries, and is indexed with the 12 least significant bits of the branch PC. Each entry of table 0 contains a 3-bit up-down saturating counter, and a bit m (m stands for *meta-predictor*) which function is described in Section 3. Table 0 uses a total of $4k \times (3+1) = 16$ Kbits of storage.

The 4 other tables are indexed both with the branch PC and some global history bits : tables 1,2,3 and 4 are indexed respectively with the 10,20,40 and 80 most recent bits in the 80-bit global history, as indicated on Figure 1. When the number of global history bits exceeds the number of index bits, the global history is "folded" by a bit-wise XOR of groups of consecutive history bits, then it is XORed with the branch PC as in a gshare predictor [4]. For example, table 3 is indexed with 40 history bits, and the index may be implemented as $pc[0:9] \oplus h[0:9] \oplus h[10:19] \oplus h[20:29] \oplus h[30:39]$ where $\oplus$ denotes the bit-wise XOR. Section 4 describes precisely the index functions that were used for the submission. Each of the tables 1 to 4 has 1k entries. Each entry contains a 8-bit tag, a 3-bit up-down saturating counter, and a bit u (u stands for "useful entry", its function is described in Section 3), for a total of 12 bits per entry. So each of the tables 1 to 4 uses $1k \times (3+8+1) = 12$ Kbits.

The total storage used by the predictor is $16k + 4 \times 12k = 64$ Kbits.

2 Obtaining a prediction

At prediction time, the 5 tables are accessed simultaneously. While accessing the tables, a 8-bit tag is computed for each table 1 to 4. The hash function used to compute the 8-bit tag is different from the one used to index the table, but it takes as input the same PC and global history bits.

Once the access is done, we obtain four 8-bits tags from tables 1 to 4, and 5 prediction bits from tables 0 to 4 (the prediction bit is the most significant bit of the 3-bit counter). We obtain a total of $4 \times 8 + 5 = 37$ bits. These 37 bits are then reduced to a $0/1$ final prediction, which is obtained as the most significant bit of the

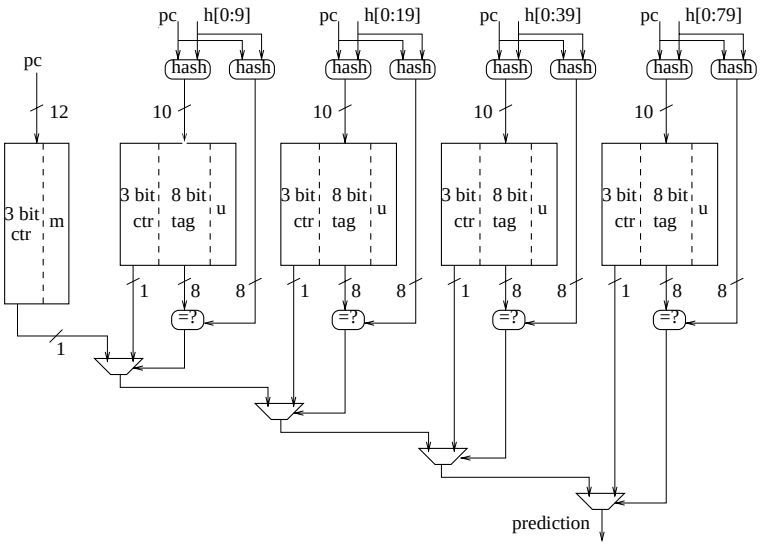

Figure 1: The proposed predictor features 5 tables. The "bimodal" table on the left has 4k entries, with 4 bits per entry. Each of the 4 other tables has 1k entries, with 12 bits per entry. The table on the right is the one using the more global history bits (80 bits).

3-bit counter associated with the longest matching history. That is, if the computed tag on table 4 matches the stored tag, we take the prediction from table 4 as the final prediction. Otherwise, if the computed tag on table 3 matches the stored tag, we take the prediction from table 3. And so on. Eventually, if there is a tag mismatch on each table 4 to 1, the final prediction is given by table 0.

3 Predictor update

At update time (for instance at pipeline retirement), we remember what was the prediction, from which table X the prediction was obtained (table $X \in [0, 4]$), and we know whether the prediction was correct or not.

Update 3-bit counter. We update the 3-bit counter on table X, the one that provided the final prediction, and only that counter. This is the classical method [7] : the counter is incremented if the branch is taken, decremented otherwise, and it saturates at values 7 and 0. In general, people prefer to use 2-bit counters instead of 3-bit counters. However, in the proposed predictor, 3-bit counters generate less mispredicts.

Allocate new entries. If $X \leq 3$, and if the prediction was wrong, we allocate one or several entries in tables $n > X$ (there is no need to allocate new entries if the prediction was correct). Actually, there was a tag miss on each table $n > X$ at prediction time. The allocation consists in "stealing" the corresponding entries by writing the computed tag for the current branch. This is done as follows. We read the $4 - X$ bits u from tables $X + 1$ to 4. If all bits u are set, we chose a random $Y \in [X + 1, 4]$ and "steal" the entry only on table Y. Otherwise, if at least one among the $4 - X$ bits u is reset, we "steal" only the entries which have their bit u reset.

As said previously, a "stolen" entry is reinitialized with the computed tag of the current branch. Moreover, the associated bit u is reset. Finally, the associated 3-bit counter is reinitialized either with value 3 (weakly not-taken) or 4 (weakly taken). This is done as follows. We read bit m from table 0. If m is set,

we reinitialize the 3-bit counter according to the branch outcome, i.e., value 4 if the branch is taken, value 3 if the branch is not taken. Otherwise, if bit m is reset, we reinitialize the 3-bit counter according to the bimodal prediction from table 0, i.e., value 3 if the bimodal prediction is *not-taken*, value 4 if the bimodal prediction is *taken*.

Updating bits u and m. If the final prediction was different from the bimodal prediction (which implies $X > 0$), we update bit u in table X and bit m in table 0 as follows. If the final prediction was correct, bits m and u are both set, otherwise they are both reset.

The rationale is as follows. If the final prediction differs from the bimodal prediction, there are two situations :

- Bimodal is wrong. It means that the entry in table X is a useful entry. By setting bit u, we indicate that we would like to prevent this entry from being stolen by another branch. By setting bit m, we indicate that the branch outcome exhibits correlation with the global history value, so new entries for that branch should be allocated by reinitializing the 3-bit counter according to the actual branch outcome.
- Bimodal is correct. Prediction from table X was wrong. This happens when a branch exhibits randomness in its behavior, and its outcome is not correlated with the global history value (or the global history is not long enough). In that case, we are allocating many useless entries. Moreover, because allocation is done only upon mispredicts, it is safer to initialize 3-bit counters with the bimodal prediction, which represents the most likely branch outcome. So we reset bit m to mean this. Moreover, we reset bit u to indicate that the entry has not yet been proven useful, so it can be stolen if another branch claims the entry.

More detailed explanations can be found in [5].

4 Folded global history

A possible way to implement history folding would be to use a tree of XOR. For example, for the 40-bit history, $h[0:9] \oplus h[10:19] \oplus h[20:29] \oplus h[30:39]$ requires a depth-2 tree (assuming 2-input XORs). For the 80-bit history, this requires a depth-3 tree.

In practice, history folding can be implemented by taking advantage of the fact that we are not folding a random value, but a global history value derived from the previous history value [6]. Figure 2 shows two examples of how global history folding can be implemented with a circular shift register (CSR) and a couple of XORs.

In the proposed predictor, we put a 10-bit CSR in front of each table 2 to 4 to compute the index. The index is then obtained by a bitwise XOR of the CSR bits with $pc[9:0] \oplus pc[19:10]$. In front of table 1, the index is directly $pc[9:0] \oplus pc[19:10] \oplus h[9:0]$.

History folding is used also for the tags. For each table 1 to 4, we use a set of two CSRs, CSR1 and CSR2, which are respectively 8 bits and 7 bits. The tag is computed as $pc[7:0] \oplus CSR1 \oplus (CSR2 << 1)$. We used two CSRs because a single CSR is sensitive to periodic patterns in the global history, which is a frequent case.

The shift register used to maintain the 80-bit global history and all the CSRs (three 10-bit CSRs, four 8-bit CSRs and four 7-bit CSRs) amount to 170 bits.

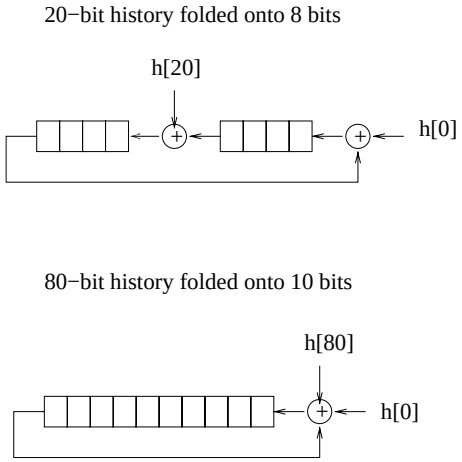

Figure 2: Global history folding can be implemented with a circular shift register (CSR) and a couple of XORs (symbol $\oplus$).

References

[1] I.-C.K. Chen, J.T. Coffey, and T.N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[2] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[3] A.N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.

[4] S. McFarling. Combining branch predictors. Technical note TN-36, DEC WRL, June 1993.

[5] P. Michaud. Analysis of a tag-based branch predictor. PI-1660, IRISA, November 2004.

[6] P. Michaud and A. Seznec. A comprehensive study of dynamic global-history branch prediction. PI-1406, IRISA, June 2001.

[7] J.E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.