Simulation and Analysis of Two-Level Branch Prediction

Tyler Paul

CS 590 - Advanced Computer Architecture

1 Abstract

Due to modern computer processors being pipelined, branch prediction techniques are necessary to avoid pipeline stalls caused by not knowing the outcome of a brach. Two-level adaptive branch prediction using PAp is one approach to predict branches. This technique uses a branch history table (BHT) and pattern history tables (PHTs) (one for each entry in the BHT) to contain information about branches in order to make predictions with a high accuracy. We implement a Java program to simulate a PAp branch predictor. Through this predictor we explore the affects of the associativity of the BHT and number of branch history bits in the BHT on the prediction accuracy of the predictor. We find that a direct-mapped cache performs better in the case of the trace file *gccSmall*. Additionally, we find that increasing the number of branch history bits increases the prediction accuracy. Overall, predicition accuracies between 90% and 95% are attainable for the *gccSmall*.

2 Introduction

Modern computer processors are pipelined; that is the processor datapath is divided into several stages which allows for multiple instructions to be processed concurrently. The major advantage of pipeling is that instruction throughput can be increased by a factor of roughly the number of stages (in the absence of hazards). However, a problem arises when the processor must process conditional branch instructions. The branch target address of a branch cannot be known at the time of the next instruction fetch since we do not know ahead of time if the branch will be taken or not taken. Therefore without additional hardware of some other mechanism to remedy this problem, stalls would be required (numerous stalls in the case of deep pipelines) in order to wait for the branch target address to be calculated. One approach to this problem is predicting in hardware whether a branch will be taken or not throught a mechanism called dynamic branch prediction. In particular, in this paper we will simulate (in Java) and analyze one method of dynamic branch prediction called two-level adaptive branch prediction; a method first designed by TY Yeh.

3 Two-Level Adaptive Branch Prediction Using PAp

The particular scheme of two-level branch prediction we will analyze is called two-level adaptive branch prediction using a per-address branch history table and per address pattern history tables (denoted PAp). We now briefy discuss what this means. First of all, this two-level branch prediction scheme consists of two primary data structures; a branch history table (BHT) and a pattern history table (PHT). The branch history table acts as a cache for branch instructions. Each entry in the table consists of tag bits and a branch history shift register. This register will hold the information regarding the result of past branches.

For example if the contents of a branch history register are 001 then this indicates that the last occurence of the branch corresponding to the entry in the BHT was taken (as indicated by 1) and the 2nd and 3rd to last occurences were not taken (as indicated by 0). Furthermore, each entry in the BHT has a corresponding PHT which is indexed by the branch history register. Each entry in the PHT is the state of an automata (a 2-bit saturating up-down counter in our implementation) denoted A2 which is used to predict whether the corresponding branch is taken or not taken. Once the result of a branch is known the result is shifted into a branch history register and the PHT is updated accordingly. In summary, this two-level approach has the capacility to detect patterns of when branches are taken. This capacility allows for a high prediction accuracy. We will discuss further details in a later section.

4 Java Implementation

We can implement the branch predictor using object orient programming techniques in Java. Below is an overview of the classes used in our implementation. Please see the source code for further details.

4.1 *BHT.java*

A branch history table is a cache, so we must consider whether to implement it as a direct-mapped cache, set-associative, or fully-associative cache. In order to be as general as possible, we allow for the associativity of the cache to be specified in the constructor. Additionally either a least recently used (LRU) or random replacement policy can be specified by setting the lruReplaceEnable flag in the constructor. If we set lruReplaceEnable to be true then LRU replacement is used othewise a random replacement policy is used. The BHT consists of several entries each consisting of a branch history registers and tag bits. The main method of interest in our BHT class is the *cacheAddress* method. This method takes a branch address as a parameter and checks if the branch address is in the BHT cache. If there is a cache hit, then we simply return the index of the address in the BHT. On the other hand, if there is a cache miss then the new entry replaces an old entry (based on the replacement policy) in the BHT and the index is returned. In the case of LRU replacement we make use of a LRUQueue class (explained in more detail below) to determine which entry to replace. Another important method is the *updateHistory* method. This method left shifts a branch result into a history register of a specified entry in the BHT.

4.2 *LRUQueue.java*

The LRUQueue is a queue data structure (implemented by a linked list) used for a LRU replacement policy. The main methods of interest are the *update* and *dequeue* methods. The *update* method causes an entry to be removed from the queue and reinserted into the front of the queue. This method is used to indicate that a particular entry in a set in an associative cache has recently been used. Thus when *dequeue* is called then the least recently used index will be returned since it will be at the back of the queue. The possible values that the queue can hold are values between 0 and the associativity of the BHT. A value will be an index of a branch address (which we call an offset) in the associative BHT in a particular set. The actual index of the branch instruction in the BHT will be given be $setNumber * associativity + offset$.

4.3 *PHT.java*

The PHT class implements a pattern history table. This class simply contains an array of states for an automata. The array is indexed by the history register for a given entry in the BHT.

4.4 *PPHT.java*

The PPHT class implements a per address pattern history table. It basically encapsulates all of the individual PHTs corresponding to entries in the BHT. The class offers methods to easily get predictions from and update a specified PHT.

4.5 *A2.java*

The A2 class implements the 2-bit saturating up-down counter. There *getNextState* method returns the new state of the automata given an old state and a branch result. This new state can then be stored in a PHT. The *getPrediction* method give the prediction of a branch based on the state of the automata.

4.6 *TwoLevelBranchPredictorPAp.java*

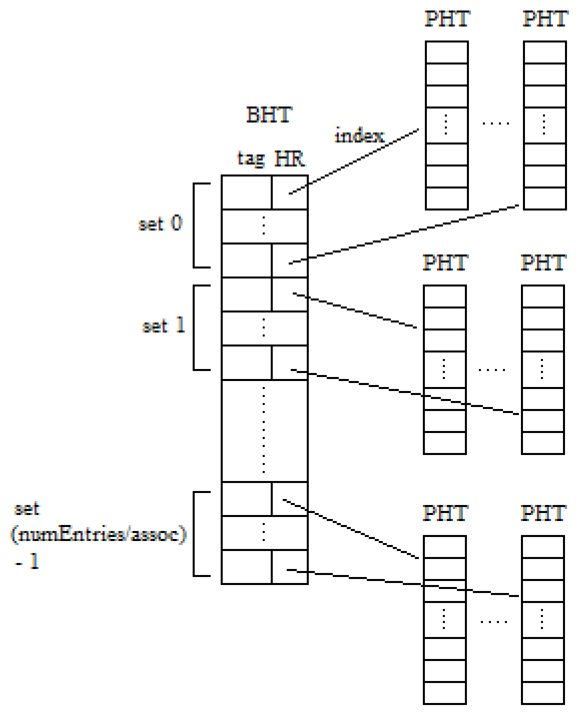
The TwoLevelBranchPredictorPAp class consists of a BHT and a PPHT which is used to make predictions of branches. The constructor takes in a file name as a parameter and reads the file line by line and processes each branch instruction (via the *processInstruction* method) and keeps track of the number of correct predictions. The *processInstruction* method operates in several steps. It first caches the branch instruction in the BHT (if it isn't already) and gets the index of it in the BHT. Next, the history bits of the BHT entry is read. Next, we index into the corresponding PHT according to the history register bits. We get the prediction and check if our prediction matches the actual branch result. Finally, we update the BHT history bits and update the PHT.

4.7 *TwoLevelBranchPredictorPApTester.java*

This class simply allows the user to set the parameters for the branch predictor, constructs the branch predictor, and prints out the results to standard output.

5 Overall Data Structure

The relationship between the BHT and PHTs can be seen in the diagram below. Note that each entry in the BHT has a corresponding PHT indexed by its history register contents. Additionally, the BHT consists of several sets depending on the associativity of the cache. If the associativity is 1, then each set consists of only one entry; i.e. a direct mapped cache. A prediction for a branch can be obtained by using the pattern history bits of an entry as input to the A2 automata.



6 Simulation Results and Analysis

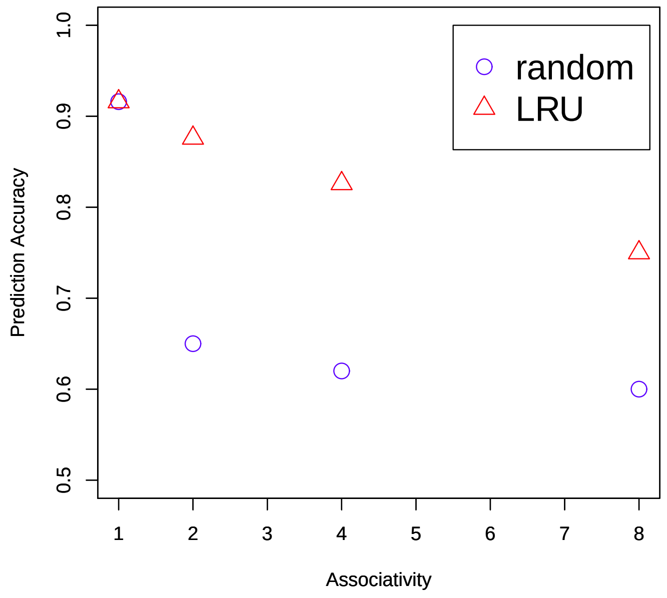
The file *gccSmall.trace*, only a portion of the full gcc program, was used to simulate our branch predictor. The file contains 2,684,455 branch addresses and a symbol indicating if each branch was taken or not ("+" for taken and "-" for not taken). The results for a 512 entry BHT with 10 history bits is shown below when both a LRU and random replacment policy is used.

Associativity	Hit %	Prediction %
1	41.7	91.6
2	42.6	87.7
4	42.7	82.6
8	42.8	75.4
16	42.6	68.8
32	42.3	64.1
64	42.5	61.0
128	52.0	62.0
256	58.4	62.6
512	84.1	73.4

Figure 1: LRU Replacement

Associativity	Hit %	Prediction %
1	41.7	91.6
2	42.3	64.9
4	42.4	62.0
8	42.5	60.3
16	42.4	59.3
32	42.4	58.8

Figure 2: Random Replacement



We now anlayze these results. First of all, note that the direct-mapped cache exhibits the best prediction percentage. At first glance this may seem odd. As associativity increases so should the hit percentage of the cache. A higher hit percentage would seem to reduce aliasing in the cache and thus increase prediction percentage. However, the gccSmall program has 1227 branches. For a set-associative cache, due to these large amount of branches there may be more very active branches (occurs often in the program) than there are entries in a set. This would mean that the LRU entry (which will be a very active branch) in the set will be removed from the set the next time there is a cache miss. However, since the LRU entry is very active, then it will soon try to cache into the BHT again. However, this time the entry will most likely be placed in a location in the cache different from its original location. The problem with this is that its pattern history bits will not be present since the entry is now in a different location. Therefore if a program has a large number of active threads that all contend for entries in their sets, then history information will often be lost causing set-associative caches to give a lower prediction accuracy. On the hand, for the same situation direct-mapped caches will perform better. Since a branch instruction can only map to one entry in the BHT then even though it may be replaced, when it re-enters the cache some of its pattern history information will still be present. This explains the correlation of prediction accuracy and associaitivity in our case. For a program with a smaller number of branches we would expect the prediction accuracy to increase with associativity. We can show further evidence for this by considering an extreme case. For a 2048 entry cache that is fully-associative there are only 1227 caches misses (caused by all 1227 branches originally entering the BHT) giving a hit percentage of 99.954. In this case our program gives a prediction accuracy of 95.41. This shows how associatiivity can increase hit percentage (althought this large fully associative cache is quite expensive). Also note that for the LRU replacement case associativity does increase prediction accuracy upon higher associativity levels starting with a level of 64 (i.e. prediction accuracy increases from going from 64 to 128 and so on). This indicates that the set sizes are getting large enough to hold many of the very active branches that map to a particular set, thus increasing prediction accuracy.

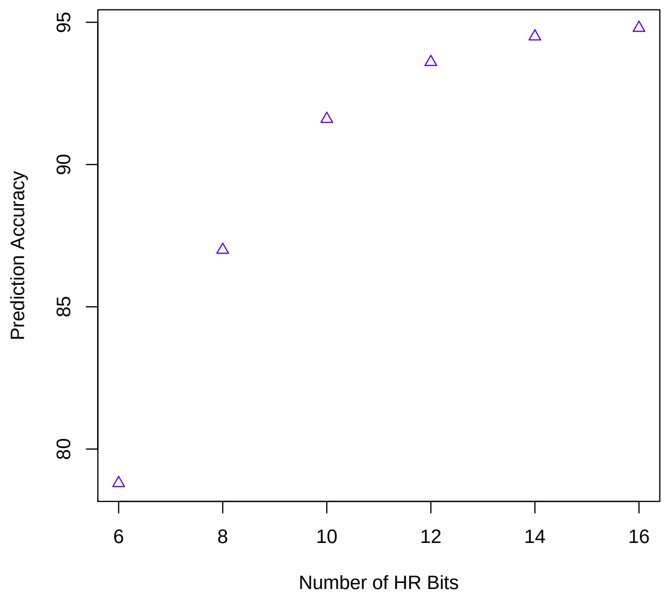
Random replacement also performs poorly for predicting gccSmall branches upon increasing associativity since recently used very active branches are often replaced in the cache upon cache misses (even more so than LRU replacement).

One approach to help cause prediction accuracy to increase with associativity would be to allow all entries in a given set to share the same PHT. This only slightly increase the prediction accuracy (by around 2%) in many of the trials of gccSmall. However, if all entries shared a PHT then if multiple branches have similar patterns but with different branch results, then there will be contention in the PHT and prediction accuracy is negatively affected.

Overall, in the case of gccSmall, a direct-mapped cache offers the best prediction accuracy since direct-mapped caches eliminate the possibility for very active branches to switch to another entry in a set in the cache (loosing patten history information).

We now explore the affect of the number of history register bits on prediction accuracy. Increasing the number of history register bits allows for more patterns of branch results to be recorded, so for this reason we would expect prediction accuracy to increase. The data below supports this claim.

Number of HR Bits	Prediction %
6	78.8
8	87.0
10	91.6
12	93.6
14	94.5
16	94.8



7 Hardware Cost

We now explore the hardware cost of the direct-mapped and set-associative branch predictor discussed above. The direct-mapped version does not require tag bits in the BHT since even if there is a cache miss there is only one entry in the BHT that can be replaced. Therefore the total cost for the storage space (expressed in number of bits) is given by

$$numEntries * numHRBits + numEntries * (numHRBits)^2 * 2$$

where numEntries is the number of entries in the BHT and numHRBits is the number of history register bits in an entry in the BHT. The first term is the number of bits needed by the BHT. The second term is the number of bits needed all of the PHTs put together. The reason for this is as follows: there is a PHT for each entry in the BHT, each PHT has $numHRBits^2$ entries since the history register of the BHT has that many combinations, and each entry consists of 2 bits to be used by the A2 automata. For example, in the case a direct-mapped 512 entry BHT with 10 HR bits the total hardware storage cost is $512 * 10 + 512 * 10^2 * 2 = 107520$ bits = 105 kilobits.

The set-associative (or fully-associative) implementation of the BHT requires tag bits. In particular, 32 - lg(numEntries) + lg(associativity) tag bits for each entry are needed since addresses are 32 bits wide. This is because the rightmost lg(numEntries) - lg(associativity) bits of an address are used to find the set of the address in the BHT, so the remaining bits must be used as a tag. Therefore the storage space cost is given by:

$$numEntries * numHRBits * numTagBits + numEntries * (numHRBits)^2 * 2$$

which is obviously more costly than the direct-mapped version.

Additionally, set-associative caches also need a large amount of hardware to determine if an address hits in the cache. For an increasing level of associativity, the hardware cost also increases. The LRU replacement policy is also very expensive to implement exactly. Approximations can be used to decrease the cost. On the other hand, random replacement is quite cheap.

8 Conclusion

We have now explore the PAp branch predictor how such a predictor could be implemented in Java. We have shown that the predictor can achieve around 90-95% prediction accuracy on the *gccSmall.trace* file using a direct-mapped cache. In particular we can achieve 91.6% prediction accuracy with a 512 entry direct mapped BHT with 10 HR bits for a hardware storage cost of 105 kilobits which we believe offers a good performance for the the cost. We have also seen that due to the large number of very active branches, a set-associative cache performs poorly except for a BHT with a large number of entries or with an extreme level of associativity (such as fully associative). Additionally, we have shown how increasing the number of histor bits and increasing the number of entries in the BHT causes prediction accuracy to increase. Lastly, we shown that a direct-mapped version of the branch predictor is cheaper than a set-associative version. This works to our advantage in the case of *gccSmall.trace* gccSmall, but for other programs with a smaller amount of active branches the set-associative version may perform better but cost more.