Dynamic Branch Prediction

• Dynamic branch prediction schemes are different from static mechanisms because they use the run-time behavior of branches to make predictions. Usually information about outcomes of previous occurrences of a given branch is used to predict the outcome of the current occurrence. Some of the dynamic mechanism are:
  – One-level or Bimodal: Uses a Branch History Table (BHT), a table of usually two-bit saturating counters which is indexed by a portion of the branch address.
  – Two-Level Adaptive Branch Prediction
  – MCFarling’s Two-Level Prediction with index sharing (gshare).
  – Path-based Correlated Branch Predictors.
  – Hybrid Predictor: Uses a combinations of two or more branch prediction mechanisms.
  – Mechanisms that try to solve the problem of aliasing, such as:
    • Gshare
    • The Agree Predictor.
    • The Skewed Branch Prediction Mechanism.
Branch Target Buffer (BTB)

- Effective branch prediction requires the target of the branch at an early pipeline stage.
- One can use additional adders to calculate the target, as soon as the branch instruction is decoded. This would mean that one has to wait until the ID stage before the target of the branch can be fetched, taken branches would be fetched with a one-cycle penalty.
- To avoid this problem one can use a Branch Target Buffer (BTB). A typical BTB is an associative memory where the addresses of branch instructions are stored together with their target addresses.
- Some designs store $n$ prediction bits as well, implementing a combined BTB and BHT.
- Instructions are fetched from the target stored in the BTB in case the branch is predicted-taken. After the branch has been resolved the BTB is updated. If a branch is encountered for the first time a new entry is created once it is resolved.
- Branch Target Instruction Cache (BTIC): A variation of BTB which caches the code of the branch target instruction instead of its address. This eliminates the need to fetch the target instruction from the instruction cache or from memory.
Basic Branch Target Buffer

FIGURE 4.22 A branch-target buffer.

- PC of instruction to fetch
- Look up
- Predicted PC
- Number of entries in branch-target buffer
- No: instruction is not predicted to be branch. Proceed normally
- Yes: then instruction is branch and predicted PC should be used as the next PC
- Branch predicted taken or untaken
One-Level Bimodal Branch Predictors

- One-level or bimodal branch prediction uses only one level of branch history.
- These mechanisms usually employ a table which is indexed by lower bits of the branch address.
- The table entry consists of n history bits, which form an n-bit automaton.
- Smith proposed such a scheme, known as the Smith algorithm, that uses a table of two-bit saturating counters.
- One rarely finds the use of more than 3 history bits in the literature.
- Two variations of this mechanism:
  - Decode History Table: Consists of directly mapped entries.
  - Branch History Table (BHT): Stores the branch address as a tag. It is associative and enables one to identify the branch instruction during IF by comparing the address of an instruction with the stored branch addresses in the table.
One-Level Bimodal Branch Predictor
Decode History Table Implementation

Branch Address

Prediction Bits
One-Level Bimodal Branch Predictor
Branch History Table Implementation
Basic Dynamic Two-Bit Branch Prediction:

Two-bit Predictor State Transition Diagram

Figure 4.13: The states in a two-bit prediction scheme.
Prediction Accuracy of A 4096-Entry Basic Dynamic Two-Bit Branch Predictor

FIGURE 4.14 Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.
From The Analysis of Static Branch Prediction:

DLX Performance Using Canceling Delay Branches

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% conditional branches</th>
<th>% conditional branches with empty slots</th>
<th>% conditional branches that are cancelling</th>
<th>% cancelling branches that are cancelled</th>
<th>% branches with cancelled delay slots</th>
<th>Total % branches with empty or cancelled delay slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>14%</td>
<td>18%</td>
<td>31%</td>
<td>43%</td>
<td>13%</td>
<td>31%</td>
</tr>
<tr>
<td>eqntott</td>
<td>24%</td>
<td>24%</td>
<td>50%</td>
<td>24%</td>
<td>12%</td>
<td>36%</td>
</tr>
<tr>
<td>espresso</td>
<td>15%</td>
<td>29%</td>
<td>19%</td>
<td>21%</td>
<td>4%</td>
<td>33%</td>
</tr>
<tr>
<td>gcc</td>
<td>15%</td>
<td>16%</td>
<td>33%</td>
<td>34%</td>
<td>11%</td>
<td>27%</td>
</tr>
<tr>
<td>li</td>
<td>15%</td>
<td>20%</td>
<td>55%</td>
<td>48%</td>
<td>26%</td>
<td>46%</td>
</tr>
<tr>
<td>Integer average</td>
<td>17%</td>
<td>21%</td>
<td>38%</td>
<td>34%</td>
<td>13%</td>
<td>35%</td>
</tr>
<tr>
<td>doduc</td>
<td>8%</td>
<td>33%</td>
<td>12%</td>
<td>62%</td>
<td>8%</td>
<td>41%</td>
</tr>
<tr>
<td>ear</td>
<td>10%</td>
<td>37%</td>
<td>36%</td>
<td>14%</td>
<td>5%</td>
<td>42%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>12%</td>
<td>0%</td>
<td>69%</td>
<td>24%</td>
<td>16%</td>
<td>17%</td>
</tr>
<tr>
<td>mdljdp2</td>
<td>9%</td>
<td>0%</td>
<td>86%</td>
<td>10%</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td>su2cor</td>
<td>3%</td>
<td>7%</td>
<td>17%</td>
<td>57%</td>
<td>10%</td>
<td>17%</td>
</tr>
<tr>
<td>FP average</td>
<td>8%</td>
<td>16%</td>
<td>44%</td>
<td>34%</td>
<td>9%</td>
<td>25%</td>
</tr>
<tr>
<td>Overall average</td>
<td>12%</td>
<td>18%</td>
<td>41%</td>
<td>34%</td>
<td>11%</td>
<td>30%</td>
</tr>
</tbody>
</table>

FIGURE 3.31 Delayed and cancelling delay branches for DLX allow branch hazards to be hidden 70% of the time on average for these 10 SPEC benchmarks.
Correlating Branches

Recent branches are possibly correlated: The behavior of recently executed branches affects prediction of current branch.

Example:

B1  if (aa==2)  aa=0;
    if (aa!=2)    SUBI  R3, R1, #2  ; b1 (aa!=2)
                   BENZ  R3, L1  ; b1 (aa!=2)
                    ADD  R1, R0, R0  ; aa==0
B2  if (bb==2)  bb=0;
    if (bb!=2)    SUBI  R3, R1, #2  ; b2 (bb!=2)
                    BNEZ  R3, L2  ; b2 (bb!=2)
                    ADD  R2, R0, R0  ; bb==0
B3  if (aa!==bb) SUB  R3, R1, R2  ; R3=aa-bb
    L1: SUBI  R3, R1, #2  
         BNEZ  R3, L2  ; b3 (aa==bb)
         ADD  R2, R0, R0  ; bb==0
         BEQZ  R3, L3  ; b3 (aa==bb)

Branch B3 is correlated with branches B1, B2. If B1, B2 are both not taken, then B3 will be taken. Using only the behavior of one branch cannot detect this behavior.
Two-Level Adaptive Predictors

- Two-level adaptive predictors were originally proposed by Yeh and Patt (1991).
- They use two levels of branch history.
- The first level stored in a Branch History Register (BHR), or Table (BHT), usually a k-bit shift register(s).
- The data in this register is used to index the second level of history, the Pattern History Table (PHT).
- Yeh and Patt later identified nine variations of this mechanism depending on how branch history and pattern history is kept: per address, globally or per set, plus they give a taxonomy (1993).
A General Two-level Predictor

Branch History Register (BHR) or Path History Register (PHR)

Pattern History Table (PHT)
- 2-bit counter
- 2-bit counter
- 2-bit counter
- 2-bit counter

High bit used to predict taken or not taken
## Taxonomy of Two-level Adaptive Branch Prediction Mechanisms

<table>
<thead>
<tr>
<th>Branch History</th>
<th>Pattern History</th>
<th>Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Globally kept</td>
<td>Globally kept</td>
<td>GAg</td>
</tr>
<tr>
<td>Globally kept</td>
<td>Kept per address</td>
<td>GAp</td>
</tr>
<tr>
<td>Globally kept</td>
<td>Kept per set</td>
<td>GAs</td>
</tr>
<tr>
<td>Kept per set</td>
<td>Globally kept</td>
<td>SAg</td>
</tr>
<tr>
<td>Kept per set</td>
<td>Kept per address</td>
<td>SAP</td>
</tr>
<tr>
<td>Kept per set</td>
<td>Kept per set</td>
<td>SAs</td>
</tr>
<tr>
<td>Kept per address</td>
<td>Globally kept</td>
<td>PAg</td>
</tr>
<tr>
<td>Kept per address</td>
<td>Kept per address</td>
<td>PAp</td>
</tr>
<tr>
<td>Kept per address</td>
<td>Kept per set</td>
<td>PAs</td>
</tr>
</tbody>
</table>
Hardware cost of Two-level Adaptive Prediction Mechanisms

- Neglecting logic cost and assuming 2-bit of pattern history for each PHT entry. The parameters are as follows:
  - $k$ is the length of the history registers,
  - $b$ is the number of BHT entries,
  - $p$ is the number of sets of branches in the PHT,
  - $s$ is the number of sets of branches in HRT.

<table>
<thead>
<tr>
<th>Scheme Name</th>
<th>History Register Length</th>
<th>Number of Pattern History Tables</th>
<th>Hardware Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAg</td>
<td>$k$</td>
<td>1</td>
<td>$k + 2 \times 2^k$</td>
</tr>
<tr>
<td>GAs</td>
<td>$k$</td>
<td>$p$</td>
<td>$k + p \times 2 \times 2^k$</td>
</tr>
<tr>
<td>GAp</td>
<td>$k$</td>
<td>$b$</td>
<td>$k + b \times 2 \times 2^k$</td>
</tr>
<tr>
<td>PAg</td>
<td>$k$</td>
<td>1</td>
<td>$b \times k + 2 \times 2^k$</td>
</tr>
<tr>
<td>PAs</td>
<td>$k$</td>
<td>$p$</td>
<td>$b \times k + p \times 2 \times 2^k$</td>
</tr>
<tr>
<td>PAp</td>
<td>$k$</td>
<td>$b$</td>
<td>$b \times k + b \times 2 \times 2^k$</td>
</tr>
<tr>
<td>SAg</td>
<td>$k$</td>
<td>$s \times 1$</td>
<td>$s \times k + s \times 2 \times 2^k$</td>
</tr>
<tr>
<td>SAs</td>
<td>$k$</td>
<td>$s \times p$</td>
<td>$s \times k + s \times p \times 2 \times 2^k$</td>
</tr>
<tr>
<td>SAp</td>
<td>$k$</td>
<td>$s \times b$</td>
<td>$s \times k + s \times b \times 2 \times 2^k$</td>
</tr>
</tbody>
</table>
Variations of global history Two-Level Adaptive Branch Prediction.
Performance of Global history schemes with different branch history lengths

- The average prediction accuracy of integer (int) and floating point (fp) programs by using global history schemes. These curves are cut off when the implementation cost exceeds 512K bits.
Performance of Global history schemes with different number of pattern history tables

![Graph showing the performance of Global history schemes with different number of pattern history tables. The x-axis represents the log of the number of pattern history tables, and the y-axis represents prediction accuracy. The graph compares various schemes such as fp, 12bit_BHR, fp, 8bit_BHR, fp, 4bit_BHR, int, 12bit_BHR, int, 8bit_BHR, and int, 4bit_BHR.]
Variations of per-address history
Two-Level Adaptive Branch Prediction

<table>
<thead>
<tr>
<th>PAg</th>
<th>PAs</th>
<th>PAp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per-addr Branch History Table (PBHT)</td>
<td>SetP(B)</td>
<td>Per-addr Branch History Table (PBHT)</td>
</tr>
<tr>
<td>Addr(B)</td>
<td>m</td>
<td>Addr(B)</td>
</tr>
<tr>
<td>k</td>
<td></td>
<td>k</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>

Global Pattern History Table (GPHT)

Per-set Pattern History Tables (SPHTs)

Addr(B)
Performance of Per-address history schemes with different branch history lengths

![Graph showing prediction accuracy vs. branch history length for different schemes.](image)
Performance of Per-address history schemes with different number of pattern history tables

![Graph showing prediction accuracy vs. log(number of pattern history tables)]
Variations of per-set history Two-Level Adaptive Branch Prediction
Performance of Per-set history schemes with different branch history lengths

![Graph showing performance of different branch history schemes.](graph.png)
Performance of Per-set history schemes with different number of pattern history tables

![Graph showing prediction accuracy vs. log(number of pattern history tables in each set) for different schemes.](image-url)
Comparison of the most effective configuration of each class of Two-Level Adaptive Branch Prediction with an implementation cost of 8K bits
Comparison of the most effective configuration of each class of Two-Level Adaptive Branch Prediction with an implementation cost of 128K bits.

Benchmark, Branch Predictor
Correlating Two-Level Dynamic GAp Branch Predictors

• Improve branch prediction by looking not only at the history of the branch in question but also at that of other branches:
  – Record the pattern of the m most recently executed branches as taken or not taken globally in BHR.
  – Use that pattern to select the proper PHT.

• In general, the notation: \((m,n)\) GAp predictor means:
  – Record last m branches to select between \(2^m\) PHTs.
  – Each table uses n-bit counters (each table entry has n bits).

• Basic two-bit Bimodal BHT is then a \((0,2)\) predictor.
GAp Predictor

- The branch history is kept globally, the pattern history per branch address.
Prediction Accuracy of Two-Bit Dynamic Predictors Under SPEC89
MCFarling's gshare Predictor

- McFarling notes (1993) that using global history information might be less efficient than simply using the address of the branch instruction, especially for small predictors.
- He suggests using both global history and branch address by hashing them together. He proposes using the XOR of global branch history and branch address since he expects that this value has more information than either one of its components.
- This gives a more uniform distribution and usage of the PHT entries reducing interference.
- The result is that this mechanism outperforms a GAp scheme by a small margin.
- This mechanism seems to use substantially less hardware, since both branch and pattern history are kept globally.
- The hardware cost for k history bits is the same as Gag: \( k + 2 \times 2^k \) bits, neglecting costs for logic.
gshare Predictor

Branch and pattern history are kept globally. History and branch address are XORed and the result is used to index the PHT.
gshare Performance

![Graph showing gshare Performance with different predictor sizes](image)
Path-Based Prediction

- Ravi Nair proposed (1995) to use the path leading to a conditional branch rather than the branch history in the first level to index the PHT.
- The global history register of a GAp scheme is replaced by a Path History Register, which encodes the addresses of the targets of the preceding \( p \) branches.
- The path history register could be organized as a \( g \) bit shift register which encodes \( q \) bits of the last \( p \) target addresses, where \( g = p \times q \).
- The hardware cost of such a mechanism is similar to that of a GAp scheme. If \( b \) branches are kept in the prediction data structure the cost is \( g + b \times 2 \times 2^g \).
- The performance of this mechanism is similar to a comparable or better than branch history schemes for the case of no context switching. For the case that there is context switching, that is, if the processor switches between multiple processes running on the system, Nair proposes flushing the prediction data structures at regular intervals to improve accuracy. In such a scenario the mechanism performs slightly better than a comparable GAp predictor.
A Path-based Prediction Mechanism

Diagram showing the relationship between branch target, path history register, and prediction bits. The diagram illustrates how branch addresses are compared against the path history to predict branch outcomes.
Hybrid or Combined Predictors

- Hybrid predictors are simply combinations of other branch prediction mechanisms.
- This approach takes into account that different mechanisms may perform best for different branch scenarios.
- McFarling (1993) presented a number of different combinations of two branch prediction mechanisms.
- He proposed to use an additional 2-bit counter array which serves to select the appropriate predictor.
- One predictor is chosen for the higher two counts, the second one for the lower two counts.
- If the first predictor is wrong and the second one is right the counter is decremented, if the first one is right and the second one is wrong, the counter is incremented. No changes are carried out if both predictors are correct or wrong.
A Generic Hybrid Predictor

Diagram showing a flowchart with:
- Branch history
- Branch address
- Predictor 1
- Predictor 2
- Predictor n
- Selection

Flowchart illustrates the interaction between these components.
MCFarling’s Combined Predictor Structure

The combined predictor contains an additional counter array with 2-bit up/down saturating counters. Which serves to select the best predictor to use. Each counter keeps track of which predictor is more accurate for the branches that share that counter. Specifically, using the notation \( P1c \) and \( P2c \) to denote whether predictors P1 and P2 are correct respectively, the counter is incremented or decremented by \( P1c - P2c \) as shown below.

<table>
<thead>
<tr>
<th>( P1c )</th>
<th>( P2c )</th>
<th>( P1c - P2c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(no change) (decrement counter) (increment counter) (no change)
MCFarling’s Combined Predictor Performance by Size

- bimodalN/gshareN+1
- local/gshare
- gselect-best
- bimodal

Conditional Branch Prediction Accuracy (%) vs. Predictor Size (bytes)
Branch Prediction Aliasing/Interference

- Aliasing/Interference occurs when different branches point to the same prediction bits.
- If the branch prediction mechanism is a one-level mechanism, k lower bits of the branch address are used to index the table. Two branches with the same lower k bits point to the same prediction bits.
- Similarly, in a PAg two-level scheme, the pattern history is indexed by the contents of history registers. If two branches have the same k history bits they will point to the same predictor entry in PHT.
- Three different cases of aliasing:
  - Constructive aliasing improves the prediction accuracy,
  - Destructive aliasing decreases the prediction accuracy, and
  - Harmless aliasing does not change the prediction accuracy.
- An alternative definition of aliasing applies the "three-C" model of cache misses to aliasing. Where aliasing is classified as three cases:
  - Compulsory aliasing occurs when a branch substream is encountered for the first time.
  - Capacity aliasing is due to the programs working set being too large to fit into the prediction data structures. Increasing the size of the data structures can reduce this effect.
  - Conflict aliasing occurs when two concurrently-active branch substreams map to the same predictor-table entry.
Interference in a Two-level Predictor

Instruction Stream

Branch A

Branch B

Pattern History Table (PHT)

Branch A’s Index 0000 0011

Branch B’s Index 0000 0011

Prediction of Branch B may be altered due to the outcome of Branch A
Interference Reduction Prediction Schemes

- The Bi-Mode Predictor, 1997.
- The Filter Mechanism, 1996.
- YAGS (Yet Another Global Scheme), 1998.
The Agree Predictor

- The Agree Predictor is a scheme that tries to deal with the problem of aliasing, proposed by Sprangle, Chappell, Alsup and Patt.
- They distinguish three approaches to counteract the interference problem:
  - Increasing predictor size to cause conflicting branches to map to different table locations.
  - Selecting a history table indexing function that distributes history states evenly among available counters.
  - Separating different classes of branches so that they do not use the same prediction scheme.
- The Agree predictor converts negative interference into positive or neutral interference by attaching a biasing bit to each branch, for instance in the BTB or instruction cache, which predicts the most likely outcome of the branch.
- The 2-bit counters of the branch prediction now predict whether or not the biasing bit is correct or not. The counters are updated after the branch has been resolved, agreement with the biasing bit leads to incrementing the counters, if there is no agreement the counter is decremented. The biasing bit could be determined by the direction the branch takes at the first occurrence.
The Agree Predictor

• The hardware cost of this mechanism is that of the two-level adaptive mechanism it is based on, plus one bit per BTB entry or entry in the instruction cache.

• Simulations show that this scheme outperforms other mechanisms, especially for smaller predictor sizes, because there is more contention than in bigger predictors.
Agree Predictor Operation

Branch History Register (BHR)

Indexing Function

Pattern History Table (PHT)

Branch Address

Biasing Bit Storage (part of BTB)

Predict taken or not taken
The Bi-Mode Predictor

- The bi-mode predictor tries to replace destructive aliasing with neutral aliasing.
- It splits the PHT table into even parts. One of the parts is the choice PHT, which is just a bimodal predictor with a slight change in the updating procedure.
- The other two parts are direction PHTs; one is a “taken” direction PHT and the other is a “not taken” direction PHT. The direction PHTs are indexed by the branch address XORed with the global history.
- When a branch is present, its address points to the choice PHT entry which in turn chooses between the “taken” direction PHT and the “not taken” direction PHT. The prediction of the direction PHT chosen by the choice PHT serves as the prediction. Only the direction PHT chosen by the choice PHT is updated.
- The choice PHT is normally updated too, but not if it gives a prediction contradicting the branch outcome and the direction PHT chosen gives the correct prediction.
- As a result of this scheme, branches which are biased to be taken will have their predictions in the “taken” direction PHT, and branches which are biased not to be taken will have their predictions in the “not taken” direction PHT. So at any given time most of the information stored in the “taken” direction PHT entries is “taken” and any aliasing is more likely not to be destructive.
- In contrast to the agree predictor, if the bias is incorrectly chosen the first time the branch is introduced to the BTB, it is not bound to stay that way while the branch is in the BTB and as a result pollute the direction PHTs.
- However, it does not solve the aliasing problem between instances of a branch which do not agree with the bias and instances which do.
The Bi-Mode Predictor

- address
- history

+ choice PHT

- direction PHT NT
- direction PHT T

prediction
The Skewed Branch Predictor

- The skewed branch predictor is based on the observation that most aliasing occurs not because the size of the PHT is too small, but because of a lack of associativity in the PHT (the major contributor to aliasing is conflict aliasing and not capacity aliasing).
- The best way to deal with conflict aliasing is to make the PHT set-associative, but this requires tags and is not cost-effective.
- Instead, the skewed predictor emulates associativity using a special skewing function.
- The skewed branch predictor splits the PHT into three even banks and hashes each index to a 2-bit saturating counter in each bank using a unique hashing function per bank (f1, f2 and f3).
- The prediction is made according to a majority vote among the three banks. If the prediction is wrong all three banks are updated. If the prediction is correct, only the banks that made a correct prediction will be updated (partial updating).
- The skewing function should have inter-bank dispersion. This is needed to make sure that if a branch is aliased in one bank it will not be aliased in the other two banks, so the majority vote will produce an unaliased prediction.
- The reasoning behind partial updating is that if a bank gives a misprediction while the other two give correct predictions, the bank with the misprediction probably holds information which belongs to a different branch. In order to maintain the accuracy of the other branch, this bank is not updated.
- The skewed branch predictor tries to eliminate all aliasing instances and therefore all destructive aliasing. Unlike the other methods, it tries to eliminate destructive aliasing between branch instances which obey the bias and those which do not.
The Skewed Branch Predictor

<table>
<thead>
<tr>
<th>address</th>
<th>history</th>
</tr>
</thead>
</table>

\[ f_1 \rightarrow \text{bank1} \]
\[ f_2 \rightarrow \text{bank2} \]
\[ f_3 \rightarrow \text{bank3} \]

\[ \text{majority vote} \]
\[ \downarrow \]
\[ \text{prediction} \]
Yet Another Global Scheme (YAGS)

• As done in the agree and bi-mode, YAGS reduces aliasing by splitting the PHT into two branch streams corresponding to biases of “taken” and “not taken”.
• However, as in the skewed branch predictor, we do not want to neglect aliasing between biased branches and their instances which do not comply with the bias.
• The motivation behind YAGS is the observation that for each branch we need to store its bias and the instances when it does not agree with it.
• A bimodal predictor is used to store the bias, as the choice predictor does in the bi-mode scheme. All we need to store in the direction PHTs are the instances when the branch does not comply with its bias.
• To identify those instances in the direction PHTs small tags (6-8 bits) are added to each entry, referring to them (PHTs) now as direction caches. These tags store the least significant bits of the branch address and they virtually eliminate aliasing between two consecutive branches.
• When a branch occurs in the instruction stream, the choice PHT is accessed. If the choice PHT indicated “taken,” the “not taken” cache is accessed to check if it is a special case where the prediction does not agree with the bias. If there is a miss in the “not taken” cache, the choice PHT is used as a prediction. If there is a hit in the “not taken” cache it supplies the prediction. A similar set of actions is taken if the choice PHT indicates “not taken,” but this time the check is done in the “taken” cache.
• The choice PHT is addressed and updated as in the bi-mode choice PHT. The “not taken” cache is updated if a prediction from it was used. It is also updated if the choice PHT is indicating “taken” and the branch outcome was “not taken.” The same happens with the “taken” cache.
• Aliasing for instances of a branch which do not agree with the branch’s bias is reduced by making the direction caches set-associative:
  – LRU replacement is used policy with one exception: an entry in the “taken” cache which indicates “not taken” will be replaced first to avoid redundant information. If an entry in the “taken” cache indicates “not taken,” this information is already in the choice PHT and therefore is redundant and can be replaced.
Performance of Four Interference Reduction Prediction Schemes

YAGS6 (6 bits in the tags).

![Graph showing prediction rate vs predictor size in K-bytes for different schemes: yags6, bimode, skew, gshare.](image)
## Processor Branch Prediction Comparison

<table>
<thead>
<tr>
<th>Processor</th>
<th>Released</th>
<th>Accuracy</th>
<th>Prediction Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyrix 6x86</td>
<td>early '96</td>
<td>ca. 85%</td>
<td>BHT associated with BTB</td>
</tr>
<tr>
<td>Cyrix 6x86MX</td>
<td>May '97</td>
<td>ca. 90%</td>
<td>BHT associated with BTB</td>
</tr>
<tr>
<td>AMD K5</td>
<td>mid '94</td>
<td>80%</td>
<td>BHT associated with I-cache</td>
</tr>
<tr>
<td>AMD K6</td>
<td>early '97</td>
<td>95%</td>
<td>2-level adaptive associated with BTIC and ALU</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>late '93</td>
<td>78%</td>
<td>BHT associated with BTB</td>
</tr>
<tr>
<td>Intel P6</td>
<td>mid '96</td>
<td>90%</td>
<td>2-level adaptive with BTB</td>
</tr>
<tr>
<td>PowerPC750</td>
<td>mid '97</td>
<td>90%</td>
<td>BHT associated with BTIC</td>
</tr>
<tr>
<td>MC68060</td>
<td>mid '94</td>
<td>90%</td>
<td>BHT associated with BTIC</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>early '97</td>
<td>95%</td>
<td>2-level adaptive associated with I-cache</td>
</tr>
<tr>
<td>HP PA8000</td>
<td>early '96</td>
<td>80%</td>
<td>BHT associated with BTB</td>
</tr>
<tr>
<td>SUN UltraSparc</td>
<td>mid '95</td>
<td>88%int</td>
<td>BHT associated with I-cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>94%FP</td>
<td></td>
</tr>
</tbody>
</table>
The Cyrix 6x86/6x86MX

- Both use a single-level 2-bit Smith algorithm BHT associated with BTB.
- BTB (512-entry for 6x86MX and 256-entry for 6x86) and the BHT (1024-entry for 6x86MX).
- The Branch Target Buffer is organized 4-way set-associative where each set contains the branch address, the branch target addresses for taken and not-taken and 2-bit branch history information.
- Unconditional branches are handled during the fetch stage by either fetching the target address in case of a BTB hit or continuing sequentially in case of a BTB miss.
- For conditional branch instructions that hit in the BTB the target address according to the history information is fetched immediately. Branch instructions that do not hit in the BTB are predicted as not taken and instruction fetching continues with the next sequential instruction.
- Whether the branch is resolved in the EX or in the WB stage determines the misprediction penalty (4 cycles for the EX and 5 cycles for the WB stage).
- Both the predicted and the unpredicted path are fetched, avoiding additional cycles for cache access when a misprediction occurs.
- Return addresses for subroutines are cached in an eight-entry return stack on which they are pushed during CALL and popped during the corresponding RET.
Intel Pentium

• Similar to 6x86, it uses a single-level 2-bit Smith algorithm BHT associated with a four way associative BTB which contains the branch history information.
• However Pentium does not fetch non-predicted targets and does not employ a return stack.
• It also does not allow multiple branches to be in flight at the same time.
• However, due to the shorter Pentium pipeline (compared with 6x86) the misprediction penalty is only three or four cycles, depending on what pipeline the branch takes.
• Like Pentium, the P6 uses a BTB that retains both branch history information and the predicted target of the branch. However the BTB of P6 has 512 entries reducing BTB misses. Since the
• The average misprediction penalty is 15 cycles. Misses in the BTB cause a significant 7 cycle penalty if the branch is backward
• To improve prediction accuracy a two-level branch history algorithm is used.
• Although the P6 has a fairly satisfactory accuracy of about 90%, the enormous misprediction penalty should lead to reduced performance. Assuming a branch every 5 instructions and 10% mispredicted branches with 15 cycles per misprediction the overall penalty resulting from mispredicted branches is 0.3 cycles per instruction. This number may be slightly lower since BTB misses take only seven cycles.
AMD K5

• The branch history information is included in the instruction cache together with the location of the target instruction within the cache. This approach is very inexpensive since no BTB is used and only the location of the target within the instruction cache rather than the full address is stored.

• This approach allows AMD to keep 1024 branches predicted. However, it could happen that the target line which is referred to in a different line of the cache has already been overwritten, or that the target address is computed and has changed between two calls of a particular branch.

• To avoid wrong target instructions to be fetched a branch unit address comparison logic is employed.

• The performance is comparable with that of Intel Pentium.
AMD K5 Instruction Cache Integrated Branch Prediction Mechanism

<table>
<thead>
<tr>
<th>Instruction Block</th>
<th>Tag</th>
<th>Jump Instruction Position</th>
<th>Target Offset</th>
<th>Target Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cache index/way select of next fetch

Instruction Block | Tag |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Instruction</td>
<td></td>
</tr>
</tbody>
</table>

Target Address
AMD K6

- Uses a two-level adaptive branch history algorithm implemented in a BHT with 8192 entries (16 times the size of the P6).
- However, the size of the BHT prevents AMD from using a BTB or even storing branch target address information in the instruction cache. Instead, the branch target addresses are calculated on-the-fly using ALUs during the decode stage. The adders calculate all possible target addresses before the instruction are fully decoded and the processor chooses which addresses are valid.
- A small branch target cache (BTC) is implemented to avoid a one cycle fetch penalty when a branch is predicted taken.
- The BTC supplies the first 16 bytes of instructions directly to the instruction buffer.
- Like the Cyrix 6x86 the K6 employs a return address stack for subroutines.
- The K6 is able to support up to 7 outstanding branches.
- With a prediction accuracy of more than 95% the K6 outperforms all other current microprocessors (except the DEC Alpha).
The K6 Instruction Buffer

- 32-K-byte Level-One Instruction Cache
- 16 Bytes
- 16 Bytes
- Branch Target Cache 16x16 Bytes
- Branch Target Address Adders
- Return Address Stack 16x16 Bytes
- 2:1
- Fetch Unit
- 16 Instruction Bytes
  plus
  16 Sets of Predictive Bits
- Instruction Buffer

EECC722 - Shaaban

#60 lec # 6 Fall 2002 9-25-2002
Motorola PowerPC 750

- A dynamic branch prediction algorithm is combined with static branch prediction which enables or disables the dynamic prediction mode and predicts the outcome of branches when the dynamic mode is disabled.

- Uses a single-level Smith algorithm 512-entry BHT and a 64-entry Branch Target Instruction Cache (BTIC), which contains the most recently used branch target instructions, typically in pairs. When an instruction fetch does not hit in the BTIC the branch target address is calculated by adders.

- The return address for subroutine calls is also calculated and stored in user-controlled special purpose registers.

- The PowerPC 750 supports up to two branches, although instructions from the second predicted instruction stream can only be fetched but not dispatched.
The HP PA 8000

- The HA PA 8000 uses static branch prediction combined with dynamic branch prediction.
- The static predictor can turn the dynamic predictor on and off on a page-by-page basis. It usually predicts forward conditional branches as not taken and backward conditional branches as taken.
- It also allows compilers to use profile based optimization and heuristic methods to communicate branch probabilities to the hardware.
- Dynamic branch prediction is implemented by a 256-entry BHT where each entry is a three bit shift register which records the outcome of the last three branches instead of saturated up and down counters. The outcome of a branch (taken or not taken) is shifted in the register as the branch instruction retires.
- To avoid a taken branch penalty of one cycle the PA 8000 is equipped with a Branch Target Address Cache (BTAC) which has 32 entries.
The HP PA 8000 Branch Prediction Algorithm
The SUN UltraSparc

- Uses a single-level BHT Smith algorithm.
- It employs a static prediction which is used to initialize the state machine (saturated up and down counters).
- However, the UltraSparc maintains a large number of branch history entries (up to 2048 or every other line of the I-cache).
- To predict branch target addresses a branch following mechanism is implemented in the instruction cache. The branch following mechanism also allows several levels of speculative execution.
- The overall performance of UltraSparc is 94% for FP applications and 88% for integer applications.
The Compaq Alpha 21264

• The Alpha 21264 uses a two-level adaptive hybrid method combining two algorithms (a global history and a per-branch history scheme) and chooses the best according to the type of branch instruction encountered.

• The prediction table is associated with the lines of the instruction cache. An I-cache line contains 4 instructions along with a next line and a set predictor.

• If an I-cache line is fetched that contains a branch the next line will be fetched according to the line and set predictor. For lines containing no branches or unpredicted branches the next line predictor point simply to the next sequential cache line.

• This algorithm results in zero delay for correct predicted branches but wastes I-cache slots if the branch instruction is not in the last slot of the cache line or the target instruction is not in the first slot.

• The misprediction penalty for the alpha is 11 cycles on average and not less than 7 cycles.

• The resulting prediction accuracy is about 95% very good.

• Supports up to 6 branches in flight and employs a 32-entry return address stack for subroutines.
The Basic Alpha 21264 Pipeline

Mispredicted branch penalty: 7 cycles minimum
Possible delay in instr queue

Fetch | Transit | Map | Queue | Register | Execute | Write
--- | --- | --- | --- | --- | --- | ---
Access I-cache | Send to decoder | Rename registers | Insert in queue | Read operands | Integer calc | Write result

Register | Address | Cache1 | Cache2 | Write
--- | --- | --- | --- | ---
Read operands | Calculate address | Access D-cache | Get result, check tags | Write result

Register | FP1 | FP2 | FP3 | FP4 | Write
--- | --- | --- | --- | --- | ---
Read operands | Start FP op | FP op | FP op | Round result | Write result
Alpha 21264 Branch Prediction

Program Counter

Local History Table
1,024 × 10 bits

Local Predict
1,024 × 3 bits

Global Predict
4,096 × 2 bits

Global History

Local Prediction

Global Prediction

Choice Predict
4,096 × 2 bits

Final Prediction
The Alpha 21264 I-Cache Line