Dynamic branch prediction schemes utilize run-time behavior of branches to make predications. Usually information about outcomes of previous occurrences of branches are used to predict the outcome of the current branch.

- One-level (Bimodal or non-correlating): Uses a single Pattern History Table (PHT), a table of usually two-bit saturating counters which is indexed by a portion of the branch address.
- Taxonomy of Basic Two-Level Correlating Adaptive Branch Predictors. (BP-1)
- Path-based Correlating Branch Predictors.
- Hybrid Predictor: Uses a combinations of two or more branch prediction mechanisms. (BP-2)
- Branch Prediction Aliasing/Interference.
- Mechanisms that try to solve the problem of aliasing:

1. MCFarling’s Two-Level with index sharing (gshare), 1993. (BP-2)
3. The Bi-Mode Predictor, 1997. (BP-6, BP-10)
5. YAGS (Yet Another Global Scheme), 1998. (BP-10)

For Global two-level methods

BP-10 also covers a good overview of BP 5, 6, 7
One-Level (Bimodal) Branch Predictors

Single-Level

- One-level (bimodal or non-correlating) branch prediction uses only one level of branch history.
- These mechanisms usually employ a table which is indexed by lower N or “a” bits of the branch address.
- Each table entry (or predictor) consists of \( n \) history bits, which form an \( n \)-bit automaton or saturating counters.
- Smith proposed such a scheme, known as the Smith Algorithm, that uses a table of two-bit saturating counters. (1985)
- One rarely finds the use of more than 3 history bits in the literature.
- Two variations of this mechanism:
  - Pattern History Table (PHT): Consists of directly mapped entries. (most common implementation)
  - 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 (similar to BTB).
One-Level Bimodal Branch Predictors

Pattern History Table (PHT)

Sometimes referred to as
Decode History Table (DHT)
or
Branch History Table (BHT)

Or “a” low bits

N Low Bits of Branch Address

2-bit saturating counters (predictors)

High bit determines branch prediction
0 = NT = Not Taken
1 = T = Taken

What if two or more branches have the same low N address bits?

Table has $2^N$ entries
(also called predictors).

Example:

For $N = 12$
Table has $2^N = 2^{12}$ entries
$= 4096 = 4k$ entries

Number of bits needed = $2 \times 4k = 8k$ bits

What if different branches map to the same predictor (counter)?
This is called branch address aliasing and leads to interference with current branch prediction by other branches and may lower branch prediction accuracy for programs with aliasing.

Update counter after branch is resolved:
- Increment counter used if branch is taken
- Decrement counter used if branch is not taken

EECC722 - Shaaban

#3 lec # 6 Fall 2012 9-24-2012
One-Level Bimodal Branch Predictors

Branch History Table (BHT)

<table>
<thead>
<tr>
<th>N Low Bits of Branch Address</th>
<th>2-bit saturating counters (predictors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Tag</td>
<td>High bit determines branch prediction</td>
</tr>
<tr>
<td></td>
<td>0 = NT = Not Taken</td>
</tr>
<tr>
<td></td>
<td>1 = T = Taken</td>
</tr>
<tr>
<td></td>
<td>0 0 Not Taken</td>
</tr>
<tr>
<td></td>
<td>0 1 (NT)</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
</tr>
<tr>
<td></td>
<td>1 1 Taken</td>
</tr>
</tbody>
</table>

Not a common one-level implementation
(May lower impact of branch address aliasing (interference) on prediction performance)

From 551
Basic Dynamic Two-Bit Branch Prediction:

From 551

Two-bit Predictor State Transition Diagram (in textbook):

| 0 0 | Not Taken (NT) |
| 0 1 |
| 1 0 | Taken (T) |
| 1 1 |

Or Two-bit saturating counter predictor state transition diagram (Smith Algorithm):

The two-bit predictor used is updated after the branch is resolved
Prediction Accuracy of A 4096-Entry Basic One-Level Dynamic Two-Bit Branch Predictor

FP

SPEC89 benchmarks

Integer

<table>
<thead>
<tr>
<th>SPEC89 benchmarks</th>
<th>Frequency of mispredictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>nasa7</td>
<td>1%</td>
</tr>
<tr>
<td>matrix300</td>
<td>0%</td>
</tr>
<tr>
<td>tomcatv</td>
<td>1%</td>
</tr>
<tr>
<td>doduc</td>
<td>5%</td>
</tr>
<tr>
<td>spice</td>
<td>9%</td>
</tr>
<tr>
<td>fpppp</td>
<td>9%</td>
</tr>
<tr>
<td>gcc</td>
<td>12%</td>
</tr>
<tr>
<td>espresso</td>
<td>5%</td>
</tr>
<tr>
<td>eqntott</td>
<td>18%</td>
</tr>
<tr>
<td>li</td>
<td>10%</td>
</tr>
</tbody>
</table>

Has, more branches involved in IF-Then-Else constructs the FP (potentially correlated branched)

N=12
$2^N = 4096$

Integer average 11%
FP average 4%

(Lower misprediction rate due to more loops)

Prediction accuracy of a 4096-entry two-bit prediction buffer for the SPEC89 benchmarks.
Correlating Branches

Recent branches are possibly correlated: The behavior of recently executed branches affects prediction of current branch. Occur in branches used to implement if-then-else constructs Which are usually more common in integer than floating point code

Example:

B1  if (aa==2)          DSUBUI R3, R1, #2 ; R3 = R1 - 2
    aa=0; (not taken)   BNEZ  R3, L1 ; B1 (aa!=2)

B2  if (bb==2)          DSUBUI R3, R2, #2 ; R3 = R2 - 2
    bb=0; (not taken)   BNEZ  R3, L2 ; B2 (bb!=2)
    L1:               DADD R1, R0, R0 ; aa==0
                      BNEZ R3, L2 ; B2 (bb!=2)
    L2:               DADD R2, R0, R0 ; bb==0
                      BNEZ R3, L3 ; B3 (aa==bb)
    \                B3 taken if aa=bb

B3  if (aa!==bb){       DSUBUI R3, R1, R2 ; R3=aa-bb
    \ (not taken)      BEQZ  R3, L3 ; B3 (aa==bb)
                      B3 in this case

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.

From 551
Two-Level Correlating Adaptive Predictors

- Two-level correlating adaptive predictors were originally proposed by Yeh and Patt (1991).
- They use two levels of branch history to account for the behavior of correlated branches in the supplied prediction.

First Level:
- The first level stored in a Branch History Register (BHR), or Table (BHT), usually one or more k-bit shift register(s).

Second Level:
- The data in this register or table is used to index the second level of history, the Pattern History Table (PHT) or tables (PHTs) using an indexing function.
- Yeh and Patt later (1993) identified nine variations (taxonomy) of this mechanism depending on how branch history in the first level and pattern history in the second level are kept:

1. Globally, \( g \)
2. Per address, or \( p \)
3. Per set (index bits), \( s \)
A General Two-level Branch Predictor
(or Generic)

First Level:
Register (BHR) or Table (BHT)

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

Last branch outcome

Other Inputs

Examples of other inputs:

Per Address First Level:
Low "a" bits of branch address (per address first level)

Per Set First Level (with s=4 sets):
i index bits of branch address used to divide branches into s = 2^i sets
e.g. In paper #1 (BP-1) i=2 index bits 10, 11 used as branch set index to divide branches into s=4 sets and select between four BHRs in first level

Branch address

Set index bits

11 10

Examples of indexing functions:
- Concatenate low address bits and BHR (GAp)
- XOR low address bits and BHR (Gshare)

What if the indexing function for two or more branches map to the same predictor in the second level (this is called index aliasing)?

Index Aliasing
Taxonomy of Two-level Adaptive Branch Prediction Mechanisms

<table>
<thead>
<tr>
<th>First Level</th>
<th>Second Level</th>
<th>Designation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch History</td>
<td>Pattern History</td>
<td></td>
</tr>
<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>

Per Set First Level:

- e.g. two index bits 10, 11 (4 sets)
- i index bits of branch address used to divide branches into \( s = 2^i \) sets
- e.g. In paper #1 (BP-1) \( i=2 \) index bits 10, 11 used as branch set index to divide branches into \( s = 4 \) sets and select between four BHRs in first level

Per Address First Level:

- “a” low bits of branch address used to select from \( b = 2^a \) BHT entries
# Hardware Cost of Two-level Adaptive Prediction Mechanisms

**Storage requirements in bits**

- Neglecting logic cost, branch targets and assuming 2-bit (predictor) of pattern history for each PHT entry. The parameters are as follows:
  - \( k \) is the length of the history registers BHRs,
  - \( b \) is the number of BHT entries, and or number of PHTs in per address schemes
    - \( b = 2^a \) where \( a \) is the number of low branch address used
    - \( s = 2^i \) (i number of set index bits) is the number of sets of branches in first level (\# of BHRs).
    - \( p = 2^m \) (m number of set index bits) is the number of sets of branches in second level (\# of PHTs).

## In First Level

<table>
<thead>
<tr>
<th>Scheme Name</th>
<th>History Register Length</th>
<th>Number of Pattern History Tables (PHTs)</th>
<th>Hardware Cost (For both levels in bits)</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>

Each PHT has \( 2^k \) entries (each 2 bit predictor). Thus each PHT requires \( 2 \times 2^k \) bits.
Variations of global history Two-Level Adaptive Branch Prediction

First Level: Global, One BHR of size k bits,  Second Level: One or more PHT(s) each of size $= 2^k$ predictors

GAx : Global schemes, **first level is global**, one BHR of size k bits
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 using two levels of branch history.
- Uses two levels of branch history:

1. **First level (global):**
   - Record the global pattern or history of the k most recently executed branches as taken or not taken. Usually a k-bit shift register.  

2. **Second level (per branch address):**
   - \( b = 2^a \) pattern history tables (PHTs), each table entry has \( n \) bit saturating counter (usually \( n = 2 \) bits).
   - The low “\( a \)” bits of the branch address are used to select the proper branch prediction table (PHT) in the second level.
   - The k-bit global branch history register (BHR) from first level is used to select the correct prediction entry (predictor) within a the selected table,
   - Thus each of the \( b = 2^a \) tables has \( 2^k \) entries and each entry (predictor) is usually a 2-bit saturating counter.
   - Total number of bits needed for second level = \( 2^a \times 2 \times 2^k = b \times 2 \times 2^k \) bits + \( k \)
- In general, the notation: \( \text{GAp} (k,n) \) predictor means:
  - Record last \( k \) branches in first level BHR.
  - Each second level table (PHT) uses \( n \)-bit counters (each table entry has \( n \) bits).
- Basic two-bit single-level Bimodal BHT is then a (0,2) predictor (\( k=0 \), no BHR).
Organization of a Correlating Two-level GAp (2,2) Branch Predictor

In this Example:

- \( a = 4 \) (number of low bits of branch address used)
- \( b = 2^a = 2^4 = 16 \) (tables (PHTs) in second level)
- \( k = 2 \) (branches tracked in first level (BHR))
- \( n = 2 \) (number of predictors size in PHTs)

High bit determines branch prediction
- 0 = Not Taken
- 1 = Taken

Low 4 bits of address selects correct entry (predictor) in selected PHT

First Level
- Branch History Register (BHR) (2 bit shift register)

Second Level
- (16 PHTs)
- \( 16 = 2^a = 2^4 \)

In this Example:
- \( a = \# \text{ of low bits of branch address used} = 4 \)
- Thus: \( b = 2^a = 2^4 = 16 \) tables (PHTs) in second level
- \( k = \# \text{ of branches tracked in first level (BHR)} = 2 \)
- Thus: each table (PHT) in 2nd level has \( 2^k = 2^2 = 4 \) entries (i.e. each PHT size = 4 predictors)
- \( n = \# \text{ predictor size in PHTs} = 2 \)

Number of bits for 2nd level = \( 2^a \times 2 \times 2^k \)
- \( = 16 \times 2 \times 4 = 128 \) bits

A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.
Prediction Accuracy of Two-Bit Dynamic Predictors Under SPEC89
Branch Prediction Aliasing/Interference

- **Aliasing** occurs when different branches point to (and use) the same prediction bits (predictor) this leads to interference with current branch prediction by other branches.

- **Address Aliasing**: If the branch prediction mechanism is a one-level mechanism, lower bits of the branch address are used to index the table. Two branches with the same lower “a” bits point to the same prediction bits.

- **Index Aliasing**: Occurs in two-level schemes when two or more branches have the same index into second level.
  - For example: In a PAg two-level scheme, the pattern history (second level) is indexed by the contents of history registers (BHRs). If two branches have the same k history bits (same BHR value or index) they will point to the same predictor entry in the global PHT.

- **Three different cases of aliasing/interference based on impact on prediction:**
  1. **Constructive aliasing** improves the prediction accuracy,
  2. **Destructive aliasing** decreases the prediction accuracy, and
  3. **Harmless (or neutral) aliasing** does not change the prediction accuracy.

- An alternative classification of aliasing applies the "three-C" model of cache misses to aliasing/misprediction. Where aliasing is classified as three cases:
  1. **Compulsory aliasing/misprediction** occurs when a branch substream is encountered for the first time.
  2. **Capacity aliasing/misprediction** 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.
  3. **Conflict aliasing/misprediction** occurs when two concurrently-active branch substreams map to the same predictor-table entry.
Index Aliasing/Interference in a Two-level Predictor

- **Index Aliasing** occurs when different branches point to (and use) the same prediction bits (predictor):
  - i.e., two or more branches have the same index into second level.
- This leads to interference with current branch prediction by other branches.

Index Aliasing Example: Here branches A, B have the same index into second level

**Aliasing (address or index) → Interference**

- **Index Aliasing** occurs when different branches point to (and use) the same prediction bits (predictor):
  - i.e., two or more branches have the same index into second level.

- **Interference** leads to interference with current branch prediction by other branches.

**Aliased Index:** Both branches A and B have the same index to level 2

- **Branch A’s Index:** 0000 0011
- **Aliased Index:** Both branches A and B have the same index to level 2
- **Branch B’s Index:** 0000 0011

**Pattern History Table (PHT):**

- **2-bit counter**
- **2-bit counter**
- **2-bit counter**
- **2-bit counter**

**Chosen Predictor:**

- For both branches
- Prediction of Branch B may be altered due to the outcome of Branch A

**i.e Interference**

- (due to index aliasing)

**Index to second level PHT formed from branch address, branch history**

**same index (aliasing)**

- for two or more branches
Performance of Global history schemes with different branch history (BHR) lengths (k = 2-18 bits)

- 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.

Number of PHTs = b = 2^a
Where a is the number of low branch address or index bits used

Size of each PHT = 2^k
Where k is the size of the BHR

For 9 SPEC89 benchmarks
Performance of Global history schemes with different number of pattern history tables

Number of PHTs = $2^a$
Here “a” was varied from 0 to 10

For 9 SPEC89 benchmarks

These curves are cut off when the implementation cost exceeds 512K bits.

Number of PHTs = b = $2^a$
Where a is the number of low branch address or index bits used
Size of each PHT = $2^k$
Where k is the size of the BHR

$2^{10} = 1024$ PHTs

For 9 SPEC89 benchmarks
Most Cost-effective Global:
8K: GAs(7, 32)
128K: GAs(11, 32)

32 = 2^k = Number of PHTs
For 9 SPEC89 benchmarks

BHR length = k (here k = 7 or 11)
Notes On Global Schemes Performance

• The performance of global history schemes is sensitive to branch history register (BHR) length.
  – Using one PHT (Gag) prediction accuracy is still rising with 18-bit BHR, improving 25% when BHR is increased from 2 to 18 bits.
  – Using 16 PHTs, prediction accuracy improved 10% when BHR is increased from 2 to 18 bits.
  – Lengthening the BHR increases the probability that the history the current branch needs is in the BHR and reduces pattern history (PHT) interference.

• The performance of global history schemes is also sensitive to the number of pattern history tables (PHTs).
  – The pattern history of different branches interfere with each other if they map to the same PHT.
  – Adding more PHTs reduces the probability of pattern history interference by mapping interfering branches to different tables.

Why?

i.e. Global schemes suffer from a high level of index aliasing which results in excessive pattern history interference for short BHR length k and a small number of PHTs. Increasing BHR length and number of PHTs greatly reduces the level of index aliasing/pattern interference and thus improves prediction accuracy.

Summary of above observations
Variations of per-address history

Two-Level Adaptive Branch Prediction

First Level: Per Address, Branch History Table (BHT) with $b = 2^a$ entries, each entry is a $k$-bit BHR

Second Level: One or more PHT(s) each of size $= 2^b$ predictors

**PAg**
- Per-addr Branch History Table (PBHT)
- Global Pattern History Table (GPHT)
- Set Index bits

**PAs**
- Per-addr Branch History Table (PBHT)
- Here $m = 4$ (16 sets)
- Set Index bits

**PAp**
- Per-addr Branch History Tables (SPHTs)
- Per-addr Branch History Tables (PPHTs)

Size of BHT = $b = 2^a$ BHRs each $k$ bits

PAx: Per-Address schemes, first level is per-address, A BHT with $2^a$ BHRs (each BHR of size or length $k$ bits)
Performance of Per-address history schemes with different branch history (BHR) lengths (k= 2-18 bits)

These curves are cut of when the implementation cost exceeds 512K bits.
Performance of Per-address history schemes with different number of pattern history tables

(1-1024 PHTs)

Number of PHTs = $b = 2^a$
Where $a$ is the number of low branch address or index bits used

Size of each PHT = $2^k$
Where $k$ is the length of the BHR

These curves are cut of when the implementation cost exceeds 512K bits.
Most Cost-effective Per address:

- 8K: PAs(6, 16)
- 128K: PAs (8, 256)

Number of PHTs = \( b = 2^a \)
Where \( a \) is the number of low branch address or index bits used

Size of each PHT = \( 2^k \)
Where \( k \) is the length of the BHR

BHR length = \( k \) (here \( k = 6 \) or 8)
Number of PHTs Here (16 or 256)

\( 2^{10} = 1024 \) PHTs

Number of bits needed

BP-1
Notes On Per-Address Schemes Performance

• The prediction accuracy of per-address history schemes is not as sensitive to BHR length or number of PHTs as in global schemes
  – Using one PHT, integer prediction accuracy only improves 6% (compared to 25% for global) when BHR is increased from 2 to 18 bits.
  – Floating point performance is nearly flat when BHR length is larger than 6 bits.
  – Increasing number of PHTs results in a small performance improvement.

• Average integer performance of global schemes is higher than per-address schemes:
  Why?
  – Due to the large number of if-then-else statements in integer code which depend on the results of adjacent branches, therefore global schemes with better branch correlation perform better.

• Average floating point performance of per-address schemes is higher than in global schemes:
  Why?
  – Floating point programs contain more loop-control branches which exhibit periodic (non-correlated) branch behavior.
  – This periodic branch behavior is better retained in multiple BHRs, as a result performance is better in per-address history schemes.

i.e. Per-address schemes suffer from a lower level of index aliasing (and thus less resulting pattern history interference) for short BHR length k and a small number of PHTs than in global schemes. Thus, performance improvement from increasing BHR length and number of PHTs is less than that exhibited in global schemes.
Variations of per-set history Two-Level Adaptive Branch Prediction

**BP-1**

In paper PB-1

- \( i = 2 \), thus 4 sets in first level (BHT with 4 BHRs)

For **SAx**: Per-Set schemes, first level is per-set. A BHT with \( s = 2^i \) BHRs, \( s \) number of first-level branch sets (each BHR of size or length \( k \) bits)

- \( SAg \): Per-set Branch History Table (SBHT), \( s = 2^i \) BHRs
- \( SAs \): Global Pattern History Table (GPHT), \( p = 2^m \) SPHTs
- **SAp**: Per-set Pattern History Table (SPHTs), \( b = 2^a \) PPHTs

Here \( i = 2 \), \( s = 2^i = 4 \) sets in level 1

**Set Index Bits**

- \( SAg \) Set Index: \( i \) First-Level Set Index Bits
- \( SAs \) Set Index: \( i \) First-Level Set Index Bits
- **SAp** Set Index: \( i \) First-Level Set Index Bits
Performance of Per-set history schemes with different branch history (BHR) lengths (2-18 bits)

First level: 4 sets or 4 BHRs (set index address bits 10-11, same block of 1K address)

Number of PHTs = \( b = 2^a \)
Where \( a \) is the number of low branch addressor index bits used

Size of each PHT = \( 2^k \)
Where \( k \) is the length of the BHR

9 SPEC89 benchmarks

These curves are cut of when the implementation cost exceeds 512K bits.
Performance of **Per-set** history schemes with different number of pattern history tables

(1-1024 PHTs)

Number of PHTs = $2^a$
Here “a” was varied from 0 to 10

These curves are cut off when the implementation cost exceeds 512K bits.

i=2 set index bits 10, 11 used as branch set index to divide branches into $s=4$ sets in first level (BHT has 4 BHRs)

Number of PHTs = $b = 2^x$
Where $a$ is the number of low branch addressor index bits used

Size of each PHT = $2^k$
Where $k$ is the length of the BHR

9 SPEC89 benchmarks

$2^{10} = 1024$ PHTs
Most Cost-effective Per Set:

8K: SAs(6, 4x16)  
128K: SAs (9, 4x32)

BHR length = k (here k = 6 or 9)
Number of PHTs (here 4x16 or 4x32)

k is the branch history length.

Number of bits needed

2^{10} = 1024 PHTs

EECC722 - Shaaban
Notes On Per-Set Schemes Performance/Cost

• Increasing the BHR length improves integer prediction accuracy significantly (similar to global schemes).

• The performance of per-set schemes is less sensitive to the increase of number of PHTs than in global schemes but more sensitive than per-address schemes.

  Why?

i.e. The level of index aliasing (and resulting pattern history interference) for a small number of PHTs is lower than in global schemes but higher than per-address schemes. Thus, performance improvement from increasing the number of PHTs is less than that exhibited in global schemes but more than per-address schemes.

• Floating point performance does not improve significantly when 4x16 or 4x256 PHTs are used. This is similar to per-address schemes.

  – This is due to partitioning the branches into sets (4 sets in this study using 4 BHTs) according to their addresses (index bits).
Comparison of the most effective configuration of each class of Two-Level Adaptive Branch Prediction with an implementation cost of 8K bits

Global: GAs (7, 32)  Per-address: PAs (6, 16)  Per-set: SAs (6, 4x16)

XAx (k, # of PHTs)

Benchmark, Branch Predictor
Comparison of the most effective configuration of each class of Two-Level Adaptive Branch Prediction with an implementation cost of 128K bits

Global: GAs (11, 32)      Per-address: PAs (8, 256)      Per-set: SAs (9, 4x32)

Prediction Accuracy

XAx (k, # of PHTs)
Performance Summary of Basic Two-level Branch Prediction Schemes

• **Global history schemes perform better than other schemes on integer programs but require higher implementation costs to be effective overall**
  – Integer programs contain many if-then-else branches which are effectively predicted by global schemes due to their better correlation with other previous branches
  – However, global schemes require long BHR and or many PHTs to reduce interference for effective overall performance.

• **Per-Address history schemes perform better than other schemes on floating point programs and require lower implementation costs to be effective.**
  – Floating point programs contain many frequently-executed loop-control branches with periodic behavior which is better retained with a per-address BHT.
  – Pattern history of different branches interfere less than in global schemes, hence fewer PHTs are needed.

• **Per-set schemes have integer performance similar to global schemes. They also have floating point performance similar to per-address schemes**
  – To be effective, per-set schemes have even higher implementation costs due to the separate PHTs for each set (4 level 1 sets in this study).
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 second level PHTs.
- The global branch history register (BHR) of a GAp scheme is replaced by a Path History Register (PHR), 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 or better than a comparable branch history schemes for the case of no context switching (worse with context switching, longer warm-up time).
  - 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

First Level:

q bits each
Branch Target

p branch targets
Path History Register

Second Level:

b = 2^a PHTs
Each of size 2^g
Branch Address

Storage cost in bits for both levels (similar to Gap):
\( g + 2^a \times 2 \times 2^g \)
Hybrid Predictors
*(Also known as tournament or combined predictors)*

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

Usually only two predictors are used (i.e. n = 2) e.g. As in Alpha, IBM POWER 4 -7
The hybrid predictor contains an additional counter array (predictor selector array) with 2-bit up/down saturating counters. Which serves to select the best predictor to use. Each counter in the selector array 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 selector counter is incremented or decremented by P1c-P2c as shown.

<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 Hybrid Predictor Performance by Benchmark

Conditional Branch Prediction Accuracy (%)

- doduc
- eqntott
- espresso
- fpppp
- gcc
- li
- mat300
- nasa7
- spice
- tomcatv
- average

bimodal (Single Level)
gshare (global two-level)
bimodal/gshare (Combined)
MCFarling’s Combined Predictor Performance by Size

![Graph showing Conditional Branch Prediction Accuracy (%) vs. Predictor Size (bytes). The graph compares different prediction methods: bimodal, gshare, gselect-best, and local. The accuracy increases with predictor size, and the bimodal method shows the highest accuracy.]
The Alpha 21264 Branch Prediction

- 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

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Transit</th>
<th>Map</th>
<th>Queue</th>
<th>Register</th>
<th>Execute</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access I-cache</td>
<td>Send to decoder</td>
<td>Rename registers</td>
<td>Insert in queue</td>
<td>Read operands</td>
<td>Integer calc</td>
<td>Write result</td>
</tr>
</tbody>
</table>

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

Load-use penalty: 2 cycles minimum

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Cache1</th>
<th>Cache2</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read operands</td>
<td>Calculate address</td>
<td>Access D-cache</td>
<td>Get result, check tags</td>
<td>Write result</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>FP1</th>
<th>FP2</th>
<th>FP3</th>
<th>FP4</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read operands</td>
<td>Start FP op</td>
<td>FP op</td>
<td>FP op</td>
<td>Round result</td>
<td>Write result</td>
</tr>
</tbody>
</table>
Alpha 21264 Branch Prediction

Local History Table
1,024 × 10 bits

BHT

Pag?

Local History Table
1,024 × 3 bits

Local Predict

Global History

Gshare?
Gag?

12

Global Predict
4,096 × 2 bits

Choice
Predict
4,096 × 2 bits

Predictor Selector Array

Program Counter

Per-address

10

Final Prediction

(Also similar to IBM POWER4, 5, 6)
## Sample 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>PHT associated with BTB</td>
</tr>
<tr>
<td>Cyrix 6x86MX</td>
<td>May '97</td>
<td>ca. 90%</td>
<td>PHT associated with BTB</td>
</tr>
<tr>
<td>AMD K5</td>
<td>mid '94</td>
<td>80%</td>
<td>PHT 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>PHT 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>PHT associated with BTIC</td>
</tr>
<tr>
<td>MC68060</td>
<td>mid '94</td>
<td>90%</td>
<td>PHT associated with BTIC</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>early '97</td>
<td>95%</td>
<td>Hybrid 2-level adaptive associated with I-cache</td>
</tr>
<tr>
<td>HP PA8000</td>
<td>early '96</td>
<td>80%</td>
<td>PHT associated with BTB</td>
</tr>
<tr>
<td>SUN UltraSparc</td>
<td>mid '95</td>
<td>88% int</td>
<td>PHT associated with I-cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>94% FP</td>
<td>PHT associated with I-cache</td>
</tr>
</tbody>
</table>

S+D : Uses both static (ISA supported) and dynamic branch prediction

PHT = One Level
Interference Reduction in Global Prediction Schemes

+ address poor loop branch performance in global schemes

   - The Filter Mechanism, 1996. (BP-3)


3. The Bi-Mode Predictor, 1997. (BP-6)


5. YAGS (Yet Another Global Scheme), 1998. (BP-10)

BP-10 also covers a good overview of BP 5, 6, 7

Not covered here
1. **MCFarling's gshare Predictor**

   gshare = global history with index sharing

   - McFarling noted (1993) that using global history information might be less efficient than simply using the address of the branch instruction, especially for small predictors (why?).
   - He suggests using both **global history** (BHR) and **branch address** by hashing them together. He proposed **using the XOR of global k-bit branch history register (BHR) and k low branch address bits to index into the second level PHT**.
     - Since he expects that this indexing function to distribute pattern history states used more evenly among available predictors (in second level PHT) than just using the BHR as an index (GAg).
     - The result is that this mechanism reduces the level of index aliasing and resulting interference and thus outperforms conventional global schemes (e.g. GAg, GAp).

   - This mechanism uses less hardware than GAp, since both branch history (first level) and pattern history (second level) are kept globally (similar to GAg).
   - The hardware cost for k history (BHR) bits is \( k + 2^k \) bits, neglecting costs for logic (Cost similar to GAg).

---

**Index Function**

Why?

- **Index Function**
- Since he expects that this indexing function to distribute pattern history states used more evenly among available predictors (in second level PHT) than just using the BHR as an index (GAg).
- The result is that this mechanism reduces the level of index aliasing and resulting interference and thus outperforms conventional global schemes (e.g. GAg, GAp).

- This mechanism uses less hardware than GAp, since both branch history (first level) and pattern history (second level) are kept globally (similar to GAg).
- The hardware cost for k history (BHR) bits is \( k + 2^k \) bits, neglecting costs for logic (Cost similar to GAg).

---

**gshare is one of the most widely implemented two level dynamic branch prediction schemes**
gshare Predictor

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

Cost in bits = k + 2 x 2^k (similar to Gag)
gshare Performance

- Conditional Branch Prediction Accuracy (%)
- Predictor Size (bytes)

- gshare-best
- gselect-best
- global

SPEC89 used
The Agree Predictor

The Agree Predictor is a scheme that tries to deal with the problem of aliasing in global schemes, proposed by Sprangle, Chappell, Alsup and Patt.

They distinguish three approaches to counteract the interference problem:

1. Increasing predictor size to cause conflicting branches to map to different table locations. (i.e increase BHR bits k, and/or number of PHTs, $2^a$)
2. Selecting a history table indexing function (into second level) that distributes history states evenly among available counters (gshare does this).
3. Separating different classes of branches so that they do not use the same prediction scheme. Possible such classes: branches with tendency to be taken or not taken.

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.

In addition, the Agree predictor also offsets/improves global predictors bad loop performance.

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 1-the first occurrence, or 2-by the direction most often taken by the branch.
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 (for the agree/bias bit).
- Simulations show that this scheme outperforms gshare, especially for smaller predictor sizes, because there is more contention/interference than in bigger predictors.
Agree Predictor Operation

The 2-bit counters in PHT of the branch prediction predict whether or not the biasing bit is correct or not.

First Level (Global): One BHR
Branch History Register (BHR)

Second Level: One PHT
Pattern History Table (PHT)

<table>
<thead>
<tr>
<th>Bias or agree bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = agree</td>
</tr>
<tr>
<td>0 = disagree</td>
</tr>
</tbody>
</table>

When branch outcome:
Agrees with bias bit = increment
Disagrees with bias bit = Decrement

Additional advantage:
Offsets global predictors bad loop branch performance

BP-5
Agree Predictor Performance vs. gshare

Prediction accuracy versus predictor size for gcc.

Interference-free global predictor

Small improvement for large PHT
(Due to less interference to start with for large PHT)

- Interference-Free
- Agree
- Conventional
(gshare)

Prediction accuracy versus predictor size for gcc.

\[ K = \log_2(\text{Size of PHT}) \]
### Agree Predictor: Improvement in Accuracy from 1K to 64K Entry PHT

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Interference-Free</th>
<th>Agree</th>
<th>Conventional</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>.67%</td>
<td>5.22%</td>
<td>10.63%</td>
</tr>
<tr>
<td>go</td>
<td>4.50%</td>
<td>10.77%</td>
<td>15.77%</td>
</tr>
<tr>
<td>m88ksim</td>
<td>.13%</td>
<td>1.60%</td>
<td>3.32%</td>
</tr>
<tr>
<td>perl</td>
<td>.00%</td>
<td>1.57%</td>
<td>2.52%</td>
</tr>
<tr>
<td>vortex</td>
<td>.16%</td>
<td>2.36%</td>
<td>11.10%</td>
</tr>
<tr>
<td>xlisp</td>
<td>.60%</td>
<td>2.35%</td>
<td>5.51%</td>
</tr>
</tbody>
</table>

Agree predictor has less interference than gshare → Thus its performance less sensitive to size of PHT
Agree Predictor: Branch Biasing Bit Selection

- The biasing bit could be determined by:
  1. The direction the branch takes at the first occurrence "first time",
  2. Or by the direction most often taken by the branch "Most Often".
- Performance results indicate that the "first time" biasing bit selection method is mostly comparable to "Most Often" selection method.

Table 4: Comparison of prediction accuracies for "first time" versus "most often" biasing bit selection for gcc.

<table>
<thead>
<tr>
<th>PHT Entries</th>
<th>First Time</th>
<th>Most Often</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>87.52%</td>
<td>89.44%</td>
<td>1.92%</td>
</tr>
<tr>
<td>2K</td>
<td>88.88%</td>
<td>90.43%</td>
<td>1.55%</td>
</tr>
<tr>
<td>4K</td>
<td>89.99%</td>
<td>91.29%</td>
<td>1.30%</td>
</tr>
<tr>
<td>8K</td>
<td>90.84%</td>
<td>92.01%</td>
<td>1.17%</td>
</tr>
<tr>
<td>16K</td>
<td>91.59%</td>
<td>92.66%</td>
<td>1.07%</td>
</tr>
<tr>
<td>32K</td>
<td>92.17%</td>
<td>93.17%</td>
<td>1.00%</td>
</tr>
<tr>
<td>64K</td>
<td>92.84%</td>
<td>93.81%</td>
<td>.97%</td>
</tr>
</tbody>
</table>
The Bi-Mode Predictor

- The bi-mode predictor tries to replace destructive aliasing with neutral or constructive aliasing.
- It splits the PHT table into three parts:
  - One of the parts is the choice PHT, which is just a bimodal (single-level PHT) 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, BHR (similar to gshare).
- 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 (i.e neutral or constructive).
- 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

BHR (First Level)

Second Level:
3 PHTs

Choice PHT dynamically determines branch bias

choice PHT

indexing function to second level: XOR (similar to gshare)

x x = predictors (2-bit saturating counters)

BP-6 BP-10
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 functions.
- 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 “skewed” indexing functions used should have inter-bank dispersion (i.e different mapping/distributions in each second level bank or PHT). 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 aliased 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.
- **Disadvantages:**
  1. Redundancy of 1/3 - 2/3 of PHT size creates capacity aliasing.
  2. Slow to warm up on a context switch.

Disadvantages:

1. Redundancy of 1/3 - 2/3 of PHT size creates capacity aliasing.
2. Slow to warm up on a context switch.
The Skewed Branch Predictor

Three different “skewed” hashing (indexing) functions $f_1$, $f_2$, $f_3$ with inter-bank dispersion (i.e. different mapping/distributions) are used to index each PHT bank in the second level.

Second Level:
Three PHT banks

Update Policy:
- 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).
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.
- Additional Improvements: 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.
Direction Caches

First Level (BHR)

Second Level

YAGS

address

history

Low k-bits

k-bits

XOR

choice PHT

i.e. Bias

Similar to Bi-Mode method

T cache

NT cache

tag

2bc

2bc = Predictor = 2 bit saturating Counter

cache hit

prediction

BP-10
Performance of Four Interference Reduction Prediction Schemes

YAGS6 (6 bits in the tags).

8 SPEC95 Benchmarks used

8 SPEC95 Benchmarks used