Branch Prediction

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Reminder: Phases of Instruction Execution

- **Fetch**: Instruction bits retrieved from cache.
- **Decode**: Instructions placed in appropriate issue (aka “dispatch”) stage buffer.
- **Execute**: Instructions and operands sent to execution units. When execution completes, all results and exception flags are available.
- **Commit**: Instruction irrevocably updates architectural state (aka “graduation” or “completion”).

In order:
- PC
- I-cache
- Fetch Buffer
- Issue Buffer
- Func. Units
- Results Buffer
- Arch. State

Out of order:
- PC
- I-cache
- Fetch Buffer
- Issue Buffer
- Func. Units
- Results Buffer
- Arch. State
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?
Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?
Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?

~ Loop length x pipeline width
Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?

~ Loop length x pipeline width
Average Run-Length between Branches

Average dynamic instruction mix of SPEC CPU 2017 [Limaye and Adegbiya, ISPASS’18]:

<table>
<thead>
<tr>
<th></th>
<th>SPECint</th>
<th>SPECfp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branches</td>
<td>19 %</td>
<td>11 %</td>
</tr>
<tr>
<td>Loads</td>
<td>24 %</td>
<td>26 %</td>
</tr>
<tr>
<td>Stores</td>
<td>10 %</td>
<td>7 %</td>
</tr>
<tr>
<td>Other</td>
<td>47 %</td>
<td>56 %</td>
</tr>
</tbody>
</table>

SPECint17: perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz
SPECfp17: bwaves, cactus, lqm, wrf, pop2, imagick, nab, fotonik3d, robs

What is the average run length between branches?
Average Run-Length between Branches

Average dynamic instruction mix of SPEC CPU 2017 [Limaye and Adegbiya, ISPASS’18]:

<table>
<thead>
<tr>
<th></th>
<th>SPECInt</th>
<th>SPECfp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branches</td>
<td>19 %</td>
<td>11 %</td>
</tr>
<tr>
<td>Loads</td>
<td>24 %</td>
<td>26 %</td>
</tr>
<tr>
<td>Stores</td>
<td>10 %</td>
<td>7 %</td>
</tr>
<tr>
<td>Other</td>
<td>47 %</td>
<td>56 %</td>
</tr>
</tbody>
</table>

SPECInt17: perlbench, gcc, mcf, omnetpp, xalancbmk, x264, deepsjeng, leela, exchange2, xz
SPECfp17: bwaves, cactus, lbm, wrf, pop2, imagick, nab, fotonik3d, roms

What is the average run length between branches?
Roughly 5-10 instructions
MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
   2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*After Inst. Decode*
MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>
## MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Assuming zero detect on register read*
### MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td>After Reg. Fetch*</td>
<td></td>
</tr>
</tbody>
</table>

*Assuming zero detect on register read
MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td>After Reg. Fetch*</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>

*Assuming zero detect on register read
Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)

A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

Remainder of execute pipeline
(+ another 6 stages)
Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages
(in-order issue, 4-way superscalar, 750MHz, 2000)

Branch Target Address Known

A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

Remainder of execute pipeline (+ another 6 stages)
### Example Branch Penalties

UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

<table>
<thead>
<tr>
<th>Branch</th>
<th>Target Address Known</th>
<th>PC Generation/Mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td></td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>PF</td>
<td></td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>FB</td>
<td>Branch Address Calc/Begin Decode</td>
<td></td>
</tr>
<tr>
<td>BI</td>
<td>Complete Decode</td>
<td></td>
</tr>
<tr>
<td>BJ</td>
<td>Steer Instructions to Functional units</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td>Integer Execute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remainder of execute pipeline (+ another 6 stages)</td>
</tr>
</tbody>
</table>
Reducing Control Flow Penalty

- Software solutions

- Hardware solutions
Reducing Control Flow Penalty

- Software solutions
  - *Eliminate branches* – *loop unrolling*
    Increases run length between branches

- Hardware solutions
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches

• Hardware solutions
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches
  – *Reduce resolution time – instruction scheduling*
    Compute the branch condition as early as possible
    (of limited value)

• Hardware solutions
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches* – *loop unrolling*
    Increases run length between branches
  – *Reduce resolution time* – *instruction scheduling*
    Compute the branch condition as early as possible
    (of limited value)

• Hardware solutions
  – Bypass – usually results are used immediately
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches
  – *Reduce resolution time – instruction scheduling*
    Compute the branch condition as early as possible
    (of limited value)

• Hardware solutions
  – Bypass – usually results are used immediately
  – Change architecture – find something else to do
    *Delay slots* – replace pipeline bubbles with useful work
    (requires software cooperation)
Reducing Control Flow Penalty

• Software solutions
  – *Eliminate branches – loop unrolling*
    Increases run length between branches
  – *Reduce resolution time – instruction scheduling*
    Compute the branch condition as early as possible
    (of limited value)

• Hardware solutions
  – Bypass – usually results are used immediately
  – Change architecture – find something else to do
    *Delay slots* – replace pipeline bubbles with useful work
    (requires software cooperation)
  – *Speculate – branch prediction*
    *Speculative execution* of instructions beyond the branch
Branch Prediction

**Motivation:**

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

**Required hardware support:**

*Prediction structures:*
- Branch history tables, branch target buffers, etc.

*Mispredict recovery mechanisms:*
- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to state following branch
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

- **backward**: 90%
- **forward**: 50%
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

 ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
   bne0 (preferred taken)  beq0 (not taken)
Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:

- **backward** 90%
- **forward** 50%

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
  - `bne0 (preferred taken)`  `beq0 (not taken)`

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
  - typically reported as ~80% accurate
Dynamic Prediction

- **Input**
- **Predictor**
- **Truth/Feedback**
- **Prediction**

**Prediction as a feedback control process**

**Operations**
- Predict
- Update
Temporal correlation
The way a branch resolves may be a good predictor of the way it will resolve at the next execution

Spatial correlation
Several branches may resolve in a highly correlated manner (a preferred path of execution)
Predictor Primitive
Emer & Gloy, 1997

• Indexed table holding values

• Operations
  – Predict
  – Update

• Algebraic notation

\[
\text{Prediction} = P[\text{Width, Depth}](\text{Index}; \text{Update})
\]
One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction

1 bit

PC

A21064(PC; T) = P[ 1, 2K ](PC; T)

What happens on loop branches?
One-bit Predictor
aka Branch History Table (BHT)

Simple temporal prediction

\[ A21064(\text{PC}; T) = P[1, 2K](\text{PC}; T) \]

What happens on loop branches?

At best, mispredicts twice for every use of loop
Two-bit Predictor
Smith, 1981

- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

<table>
<thead>
<tr>
<th>On not-taken</th>
<th>On taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1 0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0 1</td>
<td>Weakly not-taken</td>
</tr>
<tr>
<td>0 0</td>
<td>Strongly not-taken</td>
</tr>
</tbody>
</table>

- Direction prediction changes only after two wrong predictions
Two-bit Predictor

Smith, 1981

- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

<table>
<thead>
<tr>
<th>On not-taken</th>
<th>On taken</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Strongly taken</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Weakly not-taken</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Strongly not-taken</td>
</tr>
</tbody>
</table>

- Direction prediction changes only after two wrong predictions

How many mispredictions per loop? ___
Two-bit Predictor
Smith, 1981

- Use two bits per entry instead of one bit
- Manage them as a saturating counter:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>On not-taken</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>On taken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td></td>
<td>Weakly taken</td>
</tr>
<tr>
<td>0 1</td>
<td></td>
<td>Weakly not-taken</td>
</tr>
<tr>
<td>0 0</td>
<td></td>
<td>Strongly not-taken</td>
</tr>
</tbody>
</table>

- Direction prediction changes only after two wrong predictions

How many mispredictions per loop? 1
Two-bit Predictor

Smith, 1981

Counter\([W,D](I; T) = P[W, D](I; \text{if } T \text{ then } P+1 \text{ else } P-1)\)

A21164(\(PC; T\)) = MSB(Counter[2, 2K](\(PC; T\)))
## Branch History Table

<table>
<thead>
<tr>
<th>Fetch PC</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Branch History Table

Fetch PC

I-Cache

Instruction

Opcode  offset

0_0
Branch History Table

Fetch PC

I-Cache

Instruction

Opcode

offset

Branch?

Target PC

0,0
Branch History Table

Fetch PC

I-Cache

Instruction

 Opcode

 offset

 Branch?

 Target PC

 $2^k$-entry BHT, 2 bits/entry

 BHT Index

 Taken/¬Taken?
Branch History Table

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
Exploiting Spatial Correlation
Yeh and Patt, 1992

if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;

If first condition false, second condition also false
Exploiting Spatial Correlation
Yeh and Patt, 1992

if (x[i] < 7) then
  y += 1;
if (x[i] < 5) then
  c -= 4;

If first condition false, second condition also false

*History register* records the direction of the last N branches executed by the processor
History Registers
aka Pattern History Table (PHT)

\[
\text{History}(PC; T) = P(PC; P \parallel T)
\]
GHist(;T) = MSB(Counter(History(0, T); T))
Global-History Predictor

\[ \text{GHist}(;T) = \text{MSB}(\text{Counter}(\text{History}(0, T); T)) \]

Can we take advantage of a pattern at a particular PC?
Local-History Predictor

\[ L\text{Hist}(PC; T) = \text{MSB}(\text{Counter}(\text{History}(PC; T); T)) \]
Local-History Predictor

Can we take advantage of the global pattern at a particular PC?

\[ \text{LHist}(PC; T) = \text{MSB}(\text{Counter}(\text{History}(PC; T); T)) \]
Global-History Predictor with Per-PC Counters

\[ \text{GHistPA}(PC; T) = \text{MSB}(\text{Counter}(\text{History}(0; T)||PC; T)) \]
Two-Level Branch Predictor
(Pentium Pro, 1995)

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)

Fetch PC

2-bit global branch history shift register

Shift in Taken/¬Taken results of each branch
Choosing Predictors

![Diagram showing the process of choosing predictors]

- LHist
- GHist
- Chooser

Output: Prediction
Choosing Predictors

Chooser = MSB(P(PC; P + (A==T) - (B==T)))

or

Chooser = MSB(P(GHist(PC; T); P + (A==T) - (B==T)))
Tournament Branch Predictor
(Alpha 21264, 1996)

- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications
TAGE predictor
Seznec & Michaud, 2006

PC

Bimodal

TAGE[L1]

TAGE[L2]

TAGE[L3]

My guess

Use me?

Final Prediction
TAGE predictor
Seznec & Michaud, 2006

\[
TAGE_{\text{TREE}}[L1, L2, L3](PC; T) = \\
TAGE[L3](PC, \\
TAGE[L2](PC, \\
TAGE[L1](PC, \text{Bimodal}(PC;T);T);T);T)
\]
TAGE component

Next Predictor

Counter

Useful

Tag

Prediction

GHist

PC

My guess

Use me?
TAGE predictor component
TAGE predictor component

\[ TAGE[L](PC, \text{NEXT}; T) = \]

\[
\begin{align*}
 idx &= \text{hash}(\text{PC}, \text{GHIST}[L](;T)) \\
\text{tag} &= \text{hash'}(\text{PC}, \text{GHIST}[L](;T)) \\
\end{align*}
\]

\[
\begin{align*}
\text{TAGE.U} &= \text{SA}(idx, \text{tag}; ((\text{TAGE} == T) && (\text{NEXT} != T))?1:\text{SA}) \\
\text{TAGE.Counter} &= \text{SA}(idx, \text{tag}; T?\text{SA}+1:\text{SA}-1) \\
\end{align*}
\]

\[
\begin{align*}
\text{use.me} &= \text{TAGE.U} && \text{isStrong(TAGE.Counter)} \\
\text{TAGE} &= \text{use.me}?\text{MSB(TAGE.Counter)}:\text{NEXT} \\
\end{align*}
\]

Notes:

SA is a set-associative structure
SA allocation occurs on mispredict (not shown)
TAGE.U cleared on global counter saturation
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

<table>
<thead>
<tr>
<th>A</th>
<th>PC Generation/Mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

Remainder of execute pipeline
(+ another 6 stages)

*UltraSPARC-III fetch pipeline*
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

Correctly predicted taken branch penalty

UltraSPARC-III fetch pipeline

A: PC Generation/Mux
P: Instruction Fetch Stage 1
F: Instruction Fetch Stage 2
B: Branch Address Calc/Begin Decode
I: Complete Decode
J: Steer Instructions to Functional units
R: Register File Read
E: Integer Execute

Remainder of execute pipeline (+ another 6 stages)
Limitations of branch predictors

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

Correctly predicted taken branch penalty

Jump Register penalty

UltraSPARC-III fetch pipeline
BP bits are stored with the predicted target address.

**IF stage:** If (BP=taken) then nPC=target else nPC=PC+4

**later:** check prediction, if wrong then kill the instruction and update BTB & BPb, else update BPb
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction =
Correct target =

⇒

Jump 100
Add .....
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target =

⇒

Instruction
Memory

Jump 100
Add .....
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

- BTB prediction = 236
- Correct target = 1032

⇒
Assume a 128-entry BTB

What will be fetched after the instruction at 1028?
- BTB prediction = 236
- Correct target = 1032

⇒ kill PC=236 and fetch PC=1032
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?

BTB prediction = 236
Correct target = 1032

⇒ kill PC=236 and fetch PC=1032

Is this a common occurrence?
Address Collisions

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?
BTB prediction = 236
Correct target = 1032

⇒ \textit{kill} PC=236 and \textit{fetch} PC=1032

\textit{Is this a common occurrence? Can we avoid these mispredictions?}
BTB is only for Control Instructions

BTB contains useful information for branch and jump instructions only
⇒ Do not update it for other instructions

For all other instructions the next PC is (PC)+4!

*How to achieve this effect without decoding the instruction?*
Keep both the branch PC and target PC in the BTB
PC+4 is fetched if match fails
Only *taken* branches and jumps held in BTB
Next PC determined *before* branch fetched and decoded
Consulting BTB Before Decoding

- The match for PC=1028 fails and 1028+4 is fetched
  ⇒ eliminates false predictions after ALU instructions

- BTB contains entries only for control transfer instructions
  ⇒ more room to store branch targets
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate

<table>
<thead>
<tr>
<th>A</th>
<th>PC Generation/Mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate

<table>
<thead>
<tr>
<th>A</th>
<th>PC Generation/Mux</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate

BHT in later pipeline stage corrects when BTB misses a predicted taken branch
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate

<table>
<thead>
<tr>
<th>Stage</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PC Generation/Mux</td>
</tr>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

*BTB/BHT only updated after branch resolves in E stage*
Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
- Dynamic function call (jump to run-time function address)
- Subroutine returns (jump to return address)
Uses of Jump Register (JR)

• Switch statements (jump to address of matching case)

• Dynamic function call (jump to run-time function address)

• Subroutine returns (jump to return address)

How well does BTB work for each of these cases?
Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

- Subroutine returns (jump to return address)

How well does BTB work for each of these cases?
Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

How well does BTB work for each of these cases?
Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)
  
  BTB works well if usually return to the same place

How well does BTB work for each of these cases?
Uses of Jump Register (JR)

- **Switch statements** (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- **Dynamic function call** (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- **Subroutine returns** (jump to return address)
  
  BTB works well if usually return to the same place
  
  ⇒ Often one function called from many distinct call sites!

*How well does BTB work for each of these cases?*
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() {
    fb();
}

fb() {
    fc();
}

fc() {
    fd();
}
```
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

\[ \text{k entries} \]

\( \text{typically k=8-16} \)
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

Push call address when function call executed

$k$ entries (typically $k=8$-$16$)
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() { fb(); }
fb() { fc(); }
fcc() { fd(); }
```

*Push call address when function call executed*

$k$ entries (typically $k=8-16$)
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

\[
\begin{align*}
fa() & \{ \ fb() ; \ } \\
fb() & \{ \ fc() ; \ } \\
fcc() & \{ \ fd() ; \ }
\end{align*}
\]

Push call address when function call executed

\( k \) entries
(typically \( k=8-16 \))
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```plaintext
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

*Push call address when function call executed*

![Diagram](Image)

- `k entries`
- `k entries (typically k=8-16)`
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

*Push call address when function call executed*

[k entries](#) (typically $k=8-16$)
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

fa() { fb(); }
fb() { fc(); }
fc() { fd(); }

Push call address when function call executed

Pop return address when subroutine return decoded

k entries
(typically k=8-16)
Line Prediction
(Alpha 21[234]64)

- For superscalar, useful to predict next cache line(s) to fetch

- Line Predictor predicts line to fetch each cycle (tight loop)
  - Untagged BTB structure – Why?
  - 21464 was to predict 2 lines per cycle

- Icache fetches block, and predictors improve target prediction
- PC Calc checks accuracy of line prediction(s)
Overview of Branch Prediction
Overview of Branch Prediction
Overview of Branch Prediction

```
| PC | Decode | Reg Read |
```
Overview of Branch Prediction

\[\text{PC} \rightarrow \text{Decode} \rightarrow \text{Reg Read}\]
Overview of Branch Prediction

PC → Decode → Reg Read → Execute
Overview of Branch Prediction

- Decode
  - Instr type, PC relative targets available
- Reg Read
- Execute
Overview of Branch Prediction

- Instr type, PC relative targets available
- Decode
- Reg Read
- Execute
Overview of Branch Prediction

- PC
- Decode: Instr type, PC relative targets available
- Reg Read: Simple conditions, register targets available
- Execute
Overview of Branch Prediction

- **PC**: Decode
  - Instr type, PC relative targets available
- **Reg Read**: Simple conditions, register targets available
- **Execute**
Overview of Branch Prediction

- Instr type, PC relative targets available
- Simple conditions, register targets available
Overview of Branch Prediction

- **Decode**: Instr type, PC relative targets available
- **Reg Read**: Simple conditions, register targets available
- **Execute**: Complex conditions available
Overview of Branch Prediction

- **PC**: Instruction type, PC relative targets available
- **Decode**: Simple conditions, register targets available
- **Reg Read**: Complex conditions available
- **Execute**:
Overview of Branch Prediction

- **PC**: Need next PC immediately
- **Decode**: Instr type, PC relative targets available
- **Reg Read**: Simple conditions, register targets available
- **Execute**: Complex conditions available
Overview of Branch Prediction

- Need next PC immediately
- Instr type, PC relative targets available
- Simple conditions, register targets available
- Complex conditions available
- Execute

BTB

Decoding

Register Read

Execute
Overview of Branch Prediction

- Need next PC immediately
- PC
- BTB
- Decode
  - Instr type, PC relative targets available
- Reg Read
  - Simple conditions, register targets available
- Execute
  - Complex conditions available

Tight loop
Overview of Branch Prediction

- PC
  - Need next PC immediately
  - Tight loop
- BTB
- Decode
  - Instr type, PC relative targets available
- Reg Read
  - Simple conditions, register targets available
- Execute
  - Complex conditions available

March 10, 2020
Overview of Branch Prediction

- **PC**: Need next PC immediately.
- **BTB**: Instr type, PC relative targets available.
- **Decode**: Simple conditions, register targets available.
- **Reg Read**: Complex conditions available.
- **Execute**: Tight loop
- **Loose loop**

March 10, 2020

MIT 6.823 Spring 2020

L10-38
Overview of Branch Prediction

- **PC**
  - Need next PC immediately
  - Tight loop

- **BTB**
  - Instr type, PC relative targets available
  - Loose loop

- **Decode**
  - Simple conditions, register targets available
  - Loose loop

- **Reg Read**
  - Complex conditions available
  - Loose loop

- **Execute**
  - Loose loop
Overview of Branch Prediction

Must speculation check always be correct?
Overview of Branch Prediction

Must speculation check always be correct?  No...

Tight loop
Loose loop
Loose loop
Loose loop
Overview of Branch Prediction

- Need next PC immediately
  - Tight loop
  - Loose loop

- Instr type, PC relative targets available
  -松散循环

- Simple conditions, register targets available
  -松散循环

- Complex conditions available
  -松散循环

Must speculation check always be correct? No...
Overview of Branch Prediction

- PC
  - Need next PC immediately
  - Tight loop

- BTB
  - Instr type, PC relative targets available
  - Loose loop

- Decode
  - Simple conditions, register targets available
  - Loose loop

- Reg Read
  - Complex conditions available
  - Loose loop

- Execute
  - Complex conditions available
  - Loose loop

Must speculation check always be correct? No...
Overview of Branch Prediction

Must speculation check always be correct?  No...

Best predictors reflect program behavior

Tight loop  Loose loop  Loose loop  Loose loop

Need next PC immediately

Instr type, PC relative targets available

Simple conditions, register targets available

Complex conditions available

BTB

PC

Decode

Reg Read

Execute
Next Lecture:
Speculative Execution
& Value Management