NOTICE: Lab Project#2 is Ready

- Available at KLMS ([http://klms.kaist.ac.kr/course/view.php?id=109155](http://klms.kaist.ac.kr/course/view.php?id=109155))

  - Includes project description and package

- Paper review (TA, mkwon@camelab.org)

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper Topic Notification to TA</td>
<td>Oct. 21 (11:30 PM)</td>
</tr>
<tr>
<td>Paper Review #1</td>
<td>Oct. 31 (11:30 PM)</td>
</tr>
<tr>
<td>Paper Critiques</td>
<td>Nov. 06</td>
</tr>
</tbody>
</table>

- Your final project proposal presentation will be held Nov. 11
Remind: Grades

- This course is project oriented, but will have two “lightweight” exams

- The midterm will have two questions (next week)
Introduction of Branch Prediction

[Quick review]: Here, we will answer the following questions:

- What is a branch?
- Why is the branch problematic from the view point of program execution?
- How does the branch prediction work?
Control Dependencies: Branch

- Branch requires considering two: **direction & target**!

### Direction
(Whether the branch is taken or not)

- Conditional branch
- Unconditional branch

### Target
(The target address if it is taken)

- Direct branch
- Indirect branch

<table>
<thead>
<tr>
<th>When/Where do the branch?</th>
<th>If condition is satisfied</th>
<th>Always branch</th>
<th>Included in instruction</th>
<th>Should refer the memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>for, while, do, until</td>
<td>goto, jump</td>
<td>goto, jump, etc..</td>
<td>Callback, procedure return, Call virt function, c# delegate</td>
</tr>
</tbody>
</table>
## Challenge: Branches are Frequent

What fraction of executed instructions are branch?

### core i7; x86 ISA (SPEC2006 INT)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Instructions</th>
<th>branch %</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>5.71E+10</td>
<td>6.9</td>
</tr>
<tr>
<td>bzip2</td>
<td>4.25E+10</td>
<td>11.1</td>
</tr>
<tr>
<td>hmmer</td>
<td>2.57E+10</td>
<td>5.3</td>
</tr>
<tr>
<td>gcc</td>
<td>6.29E+09</td>
<td>15.1</td>
</tr>
<tr>
<td>gobmk</td>
<td>8.93E+10</td>
<td>12.1</td>
</tr>
<tr>
<td>h264</td>
<td>1.09E+11</td>
<td>7.1</td>
</tr>
<tr>
<td>libquantum</td>
<td>4.18E+08</td>
<td>13.2</td>
</tr>
<tr>
<td>omnetpp</td>
<td>2.55E+09</td>
<td>16.4</td>
</tr>
<tr>
<td>perlbench</td>
<td>2.91E+09</td>
<td>17.3</td>
</tr>
<tr>
<td>sjeng</td>
<td>2.11E+10</td>
<td>14.8</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td><strong>9.4</strong></td>
</tr>
</tbody>
</table>

### ARM Coretex-A9; ARMv7 ISA (SPEC2006 FP)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Instructions</th>
<th>branch %</th>
</tr>
</thead>
<tbody>
<tr>
<td>bwaves</td>
<td>3.84E+11</td>
<td>13.5</td>
</tr>
<tr>
<td>cactusADM</td>
<td>1.02E+10</td>
<td>0.5</td>
</tr>
<tr>
<td>leslie3D</td>
<td>4.92E+10</td>
<td>6.2</td>
</tr>
<tr>
<td>milc</td>
<td>1.38E+10</td>
<td>6.5</td>
</tr>
<tr>
<td>tonto</td>
<td>1.30E+10</td>
<td>10.0</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td><strong>12.15</strong></td>
</tr>
</tbody>
</table>

Every 8th instruction is a branch

Every 10th or 11th instruction is a branch

Source: Blem et al [HPCA 2013]
Challenge: Control Flow Penalty

- Modern processors generally employ > 10 pipeline stages between the points of next PC calculation and branch resolution!

- Long pipelines
  - Branch outcome turns out after B cycles
  - No scheduling the branch until the outcome brought by

- Superscalars (e.g., 4-way)
  - Branch could be observed at every cycle or so!
  - One cycle of work, then stall the machine for ~B cycles?
Idea: But, Branch is Predictable

**Program A**

```c
for (i=0; i<100; i++)
{
    ...
}
```

**Assembly of Program A**

```
addi r10, r0, 100
addi r1, r0, r0
L1:
    ...
addi r1, r1, 1
bne r1, r10, L1
...
```

*Keep this branch to L1 until i <100*

**Program B**

```c
if (aa == 2)
    aa = 0;
if (bb == 2)
    bb = 0;
if (aa != bb)
```

**Assembly of Program B**

```
addi r2, r0, 2
bne r10, r2, L_bb
xor r10, r10, r10
L_bb:
    bne r11, r2, L_xx
    xor r11, r11, r11
L_xx:
    beq r10, r11, L_exit
...
L_exit:
```

*There's a pattern in the original code 😊*

*It looks like difficult to predict the branch direction 😐*

*Branch if the previous two branches didn't take*
Solution: Branch Prediction

- Again, we need to know two things for branch
  - Whether the branch is taken or not (direction)
  - The target address if it is taken (target)

<table>
<thead>
<tr>
<th>Case</th>
<th>Direction</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct jumps,</td>
<td>Known (always taken)</td>
<td>Easy to compute</td>
</tr>
<tr>
<td>Function calls</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditional branches</td>
<td>Difficult to predict</td>
<td>Easy to compute</td>
</tr>
<tr>
<td>(typically PC-relative)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect jumps,</td>
<td>Known (always taken)</td>
<td>Difficult to target</td>
</tr>
<tr>
<td>Function returns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Branch Prediction based Speculation

Speculative execution requires three architectural support

Support ❶ Predict the future execution stream
- Branch Predictor (both direction & target)

Support ❷ Execute the predicted instruction in O3
- Re-Order Buffer

Support ❸ Recover from branch mispredictions
- UNDO; restart the instruction fetch from the correct path
Problem: Branch Misprediction Penalty

- Example: 20 pipeline stages architecture

Single Issue (flush entailed instructions and re-fetch)

Fetch the correct path  Mispredict
Problem: Branch Misprediction Penalty

- Example: 20 pipeline stages & 8-way superscalar

The accuracy of branch prediction is important to mitigate pipeline stalls and flushes.

Now, let’s check the details of branch predictors!
Branch Prediction #1: Direction

<table>
<thead>
<tr>
<th>Case</th>
<th>Direction</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct jumps, Function calls</td>
<td>Known (always taken)</td>
<td>Easy to compute</td>
</tr>
<tr>
<td>Conditional branches (typically PC-relative)</td>
<td>Difficult to predict</td>
<td>Easy to compute</td>
</tr>
<tr>
<td>Indirect jumps, Function returns</td>
<td>Known (always taken)</td>
<td>Difficult to target</td>
</tr>
</tbody>
</table>

The direction predictor is needed to handle conditional branches (Most branches are this type!)
History of Direction Prediction

- Each branch direction predictor will be covered in order of time
- X-axis:
  - Prediction algorithm types for branch direction predictors
- Y-axis:
  - The accuracy (SPEC1992 INT) of such branch direction predictors

Branch direction predictor is mainly categorized into static/dynamic predictor

Source: MicroDesign Resources
Static Direction Prediction

- Statically predicts the branch direction at compile time
- Old processors use this scheme because of its simplicity and low cost

Accuracy on SPECint92

40%  
60%  
70%  
75%  
80%  
85%  
90%  
95%

Source: MicroDesign Resources
Static Direction Prediction

1. Always Not Taken
   - Always predict branches to be not taken
   - Easy to implement, but 30-40% accuracy

Accuracy on SPECint92

- 486
- R3000
- MicroSparc
- HyperSparc
- V800
- SH

Source: MicroDesign Resources
Static Direction Prediction

Always Taken
- Always predict branches to be taken
- 60-70% accuracy (as Always NT accuracy of 30-40%)

Accuracy on SPECint92
- 486
- R3000
- MicroSparc
- HyperSparc
- V800
- SH

Source: MicroDesign Resources
Static Direction Prediction

3 BTFNT (Backward Taken, Forward Not Taken)

- Always take the smaller address; advantage in loops
- But, doesn’t work well on programs with irregular branches

Accuracy on SPECint92

40% 45% 50% 55% 60% 65% 70% 75% 80% 85% 90% 95%

486
R3000
MicroSparc
HyperSparc
V800
SH
SuperSparc
MicroSparc-2
PA-7x00

Dynamic Predictor

Source: MicroDesign Resources
Static Direction Prediction

4 Compiler Directed

- Using techniques called profiling or feedback-directed compilation
- After the program is initially compiled, it runs using test data to determine the typical direction of each branch; the program is then recompiled to adjust the branch-prediction bits
- 75% accuracy!
Dynamic Direction Prediction

- **Static** predicts any particular branch in the same way whenever it is encountered.
- To achieve a greater accuracy, **dynamic** takes into account run-time info.
- The processor learns from its mistakes and changes its predictions to match the behavior of each particular branch.

### Accuracy on SPECint92

<table>
<thead>
<tr>
<th>Accuracy (%)</th>
<th>40%</th>
<th>60%</th>
<th>70%</th>
<th>75%</th>
<th>80%</th>
<th>85%</th>
<th>90%</th>
<th>95%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>486</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>R3000</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MicroSparc</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HyperSparc</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>V800</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SH</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SuperSparc</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MicroSparc-2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PA-7x00</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **PPC 601**
- **PPC 603**
- **Power2**
- **R4x00**
- **i960**

### Processor Types

- **Always NT**
- **Always Taken**
- **BTFNT**
- **Compiler directed**
- **Dynamic Predictor**

Source: MicroDesign Resources
Dynamic Direction Prediction

1-Bit History (Cont’d)

- Use Branch History Table (BHT) to keep a record of the previous branch behavior, allowing it to improve its predictions over time
- Maintains a single history bit for each branch
- 80% accuracy!

Accuracy on SPECint92

- 95%
- 90%
- 85%
- 80%
- 75%
- 70%
- 65%
- 60%
- 40%

The number of history-table entries in the parentheses

- Always
  - NT
  - Taken
  - BTFNT
  - Compiler directed
  - 1-bit history

Source: MicroDesign Resources
One-Bit Branch Predictor Architecture

**Branch History Table (indexing: hash-based)**

- **Hash** (32 bits) → **K bits** → **BHT** with **2^k entries (1 bit/entry)**
  - If entry 0: NT
  - If entry 1: T

**FSM for branch predictor**

- **A 2-state machine (NT or T)**
  - **FSM Update Logic**
    - When the direction is resolved, FSM updates the BHT
  - **Prev. status** → **Actual outcome**

**CAMELab**

- KAIST
One-Bit Branch Predictor Architecture

Branch History Table (indexing: hash-based)

FSM for branch predictor
Example 1: A loop using 1-bit BHT

Program A

```c
for (i=0; i<5; i++)
{
    ...
}
```

Assembly of Program A

```assembly
addi r10, r0, 5
addi r1, r1, r0
L1:
    ...
    addi r1, r1, 1
    bne r1, r10, L1
...
```

Prediction

```
0 1 1 1 1
```

Actual

```
T T T NT T T T NT T T
```

60% accuracy
Challenge: The Bit Is Not Enough

- Example: short loop (5 iterations)
  - Taken 4 times, then not taken once (exit loop)
  - First depends on the initial status of predictor
  - Not-taken mispredicted (was taken previously)

- Execute the same loop again
  - First always mispredicted (previous outcome was not taken)
  - Then 3 predicted correctly
  - Then last one mispredicted again

Each fluke/anomaly in a stable pattern results in two mispredicts per loop
Example 2: Variant Branch Cases

- Various branch cases based on length of T/NT iteration

Program A

```c
for (i=0; i<100000; i++)
{
    if ( (i%100) == 0)
        tick();
    if ( (i&1) == 1)
        odd();
}
```
**Challenge:** Not Work for Short Iter.

How often is branch outcome != previous outcome?

Program A:
```c
for (i=0; i<100000; i++)
{
    if ( (i%100) == 0)
        tick();
    if ( (i&1) == 1)
        odd();
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Branch History</th>
<th>Prediction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC08:</td>
<td>TTTTTTTTTTT ... TTTTTTTTTTT ... T</td>
<td>99.998%</td>
</tr>
<tr>
<td>DC44:</td>
<td>TTTTT ... TN TTTT ... TN TTTT ... T</td>
<td>98.0%</td>
</tr>
<tr>
<td>DC50:</td>
<td>TNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTNTN</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Direction Prediction

2-Bit History

- Branch History Table (BHT) maintains 2 bit per entry
- Now, FSM has four status (00, 01, 10, 11)
- For the best case, ~ 90% accuracy

Accuracy on SPECint92

- Always NT
- AlwaysTaken
- BTFNT
- Compiler directed
- 1-bit history
- 2-bit history

* means the target processor employs a return-address stack (will be covered later)
**Idea: Two Bits are Better Than One**

- 2-bit Saturating Up/Down Counter Predictor
  - MSB: Direction bit
  - LSB: Hysteresis bit (concept: relaxation)

```
[Actual output]
- Green arrow: If branch T
- Red arrow: If branch NT

[Prediction]
- Red dot: Predict NT
- Green dot: Predict T

ST: Strongly Taken
WT: Weakly Taken
WN: Weakly Not Taken
SN: Strongly Not Taken
```
Example 1: A loop using 2-bit BHT

Program A

```
for (i=0; i<5; i++)
{
    ...
}
```

Assembly of Program A

```
addi r10, r0, 5
addi r1, r1, r0

L1:
    ...
    addi r1, r1, 1
    bne r1, r10, L1
    ...
```

<table>
<thead>
<tr>
<th>Prediction</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
</tr>
<tr>
<td>11</td>
<td>T</td>
</tr>
<tr>
<td>11</td>
<td>NT</td>
</tr>
</tbody>
</table>

80% accuracy
1-bit History vs. 2-bit History (Example 1)

Initial training / warm-up

1-bit

0

1

1

1

2-bit

01

10

11

11

11

Only 1 mispredict per N branches now!
(Example 2: DC08: 99.999% DC44: 99.0%)
How About 0xDC50 in Example2?

Program A

```
for (i=0; i<100000; i++)
{
    if ( (i%100) == 0)
        tick();
    if ( (i&1) == 1)
        odd();
}
```

- 2-bit predictor also doesn’t predict well (50% accuracy at best)

```
DC50: SN WN SN WN SN WN SN WN SN WN SN WN ...
```

→ The accuracy is the same w/ coin toss probability!

- Fortunately, it’s still predictable
  - As the branch exhibits a repeating pattern: (NT)*
Dynamic Direction Prediction

Two Level

- Use *branch correlation*!
  - The outcome of a branch is often related to previous outcome(s)
- Use both Branch History Table (BHT) & Pattern History Table (PHT)
- 96% accuracy at the best!

Accuracy on SPECint92

- 40%
- 486
- R3000
- MicroSparc
- HyperSparc
- V800
- SH
- SuperSparc
- MicroSparc-2
- PA-7x00
- PPC 601
- PPC 603
- Power2
- R4x00
- i960
- AMD K5 (1K)
- R8000 (1K)
- 21064 (2K)
- 21066 (2K)
- 21064A (4K*)
- 21164 (2K*)
- Nx586 (2K*)
- PPC 620 (2K)
- PPC 604 (512)
- R10000 (512)
- UltraSparc (512)
- Cyrix M1 (256*)
- PA-8000 (256)
- Pentium (256)
- Pentium6

Source: MicroDesign Resources
Branch Correlations

- Can be categorized based on branch behaviors

- **Local Branch History**
  (Direction of *Branch A* given the outcomes of previous instances of *Branch A*)

- **Global Branch History**
  (Direction of *Branch Z* given the outcomes of all* previous branches A, B, ..., Y and Z)

---

**Example program**

```c
for (i=0; i<1000; i++)
{
    ...
}
```

**Example program**

```c
if (d == 0)
    b = 1;
...
if (b == 1)
    if the first branch is taken, always next branch will be taken
```
More Examples of Global Correlation

Testing same/similar conditions

void fill_foo(int* foo) {
    if (foo != NULL)  
        *foo = 23;
}

if (foo != NULL)  
    fill_foo(&foo)

- The caller might test the argument for NULL; the callee can also test foo for NULL, again

Partial correlations

if (cond1)  
    {..}
if (cond1 && cond2)  
    { ..}

- The first branch could examine cond₁, and the second branch can test cond₁ && cond₂
- If cond₁ is false, then the second branch can be predicted as false

Multiple correlations

if (cond1)  
    {..}
if (cond2)  
    {..}
if (cond1 && cond2)  
    { ..}

- The branch tests cond₁, and the second tests cond₂
- The third examines cond₁ ⊕ cond₂, which can always be predicted if the direction of first two branches is turned out
Correlated Predictors

- Track the history of a branch by using Pattern History Table (PHT)

**Branch History Register (e.g., previous Outcome)**

**Correlated Predictor (each counter if prev=0/prev=1)**

prev = 1
- prediction = N (Correct)
- prediction = N (Correct)

prev = 0
- prediction = T (Correct)
- prediction = T (Correct)
Correlated Predictors

- Track the *history* of a branch by using Pattern History Table (PHT)

- Branch History Register (e.g., previous Outcome)

- Correlated Predictor (each counter if prev=0/prev=1)

- PC (Hashed)

- prev = 1: prediction = T
  - prev = 0: prediction = T
  - prev = 1: prediction = T

- Incorrect
- Correct
- Decrease
- Increase
Deeper History Covers More Patterns

- If Pattern History Table can maintain the last 3 outcomes..
- Question: what pattern has this predictor entry learned?
  - 001 $\rightarrow$ 1; 011 $\rightarrow$ 0; 110 $\rightarrow$ 0; 100 $\rightarrow$ 1
  - 00110011001... (0011)*
Design Policy of Correlated Predictors

- As you may already be aware, there are many implementation variations for the correlated predictor concept
  1. The number of history register
     - 1 history register for all branches (global)
     - Table of history register, 1 for each branch (private)
     - Table of history registers, each shared by several branches (shared)
  2. History length (size of history register)
  3. The number of pattern history tables
- The implementation will be affected by the actual workload pattern, the size of table limits at design time being aware of accuracy demands
Dynamic Direction Prediction

Hybrid Predictor (a.k.a Tournament Predictor)

- No predictor is clearly the best
- Different branches exhibit different behaviors
- Idea: let’s have a predictor to predict which predictor will predict better
- 97% accuracy at the best!

Accuracy on SPECint92

- Always NT
- Always Taken
- BTFNT
- Compiler directed
- 1-bit history
- 2-bit history
- Two level
- Hybrid predictor

Source: MicroDesign Resources
Tournament Hybrid Predictor

- Combining two or more branch predictors
- Example:
  - Predictor 0: local, per-branch prediction, which are accessed by the PC
  - Predictor 1: correlated prediction based on the last “m” branches accessed by the global history
  - Meta-predictor of which had been the best predictor for this branch (table of 2-/3-bit counters)

FSM of meta-predictor

<table>
<thead>
<tr>
<th>Pred 0</th>
<th>Pred 1</th>
<th>Meta Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>---</td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>Inc</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>Dec</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>---</td>
</tr>
</tbody>
</table>

If meta-counter MSB = 0, use pred₀ else use pred₁
Common Combinations

- Combining two or more branch predictors
- Example:
  - Predictor 0: local, per-branch prediction, which are accessed by the PC
  - Predictor 1: correlated prediction based on the last “m” branches accessed by the global history
  - Meta-predictor of which had been the best predictor for this branch (table of 2-/3-bit counters)

1. Global history + Local history
2. “Easy” branches (2-bit history) + Global history
3. Short history + Long history

<table>
<thead>
<tr>
<th>Pred 0</th>
<th>Pred 1</th>
<th>Meta Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>---</td>
</tr>
<tr>
<td>×</td>
<td>✓</td>
<td>Inc</td>
</tr>
<tr>
<td>✓</td>
<td>×</td>
<td>Dec</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>---</td>
</tr>
</tbody>
</table>

FSM of meta-predictor

If meta-counter MSB = 0, use pred₀ else use pred₁
Branch Prediction #2: Target

<table>
<thead>
<tr>
<th>Case</th>
<th>Direction</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct jumps, Function calls</td>
<td>Known (always taken)</td>
<td>Easy to compute</td>
</tr>
<tr>
<td>Conditional branches (typically PC-relative)</td>
<td>Difficult to predict</td>
<td>Easy to compute</td>
</tr>
<tr>
<td>Indirect jumps, Function returns</td>
<td>Known (always taken)</td>
<td>Difficult to target</td>
</tr>
</tbody>
</table>

Again, target predictor is needed for indirect jumps & function returns
Let’s Predict Branch Target Address

**IF stage**: needs to know the fetch address per cycle

**ID stage**: even easily-computed branch targets need to wait until the instruction decoded or direction predicted

**EXEC stage**: for some branches (e.g., indirect) target known only after EX stage, which is way too late

So, we have a quick-and-dirty predictor for the target that only needs the address of the branch instruction
**Solution: Branch Target Buffer (BTB)**

- A cache-like storage that records branch addresses and associated targets (indexed by the instruction address)
  - Why tag? – (Hint: desiring 1 cycle BTB lookup)
  - If there is a hit in BTB for branch predicted taken:
    - PC ← Target in BTB for branch

```
Branch PC

BTB

Tag Target Tag Target ... Tag Target

Predicted Branch Direction

Branch Target
```
Challenge: Function Returns

- Function *returns* are frequent, yet
  - Address is difficult to compute (have to wait until EX stage done to know it)
  - Address is difficult to predict with BTB (function can be called from multiple places)

- But, a return address is in practice simple to predict if it’s aware of ins. semantic

```c
foo(....)
{
  ...
  0x10001000  jal bar
  0x10001004  ...
  ...
  0x10001800  jal bar
  0x10001804  ...
  ...
  0x10001CE4  jal bar
  0x10001CE8  ...
  ...
}
```

```c
bar(....)
{
  ...
  0x1000F0E0 jal baz
  0x1000F0E4 ...
  ...
  jar $ra
}
```

```c
baz(....)
{
  ...
  jar $ra
}
```

The return address is known from the time of each call!
**Solution: Return Address Stack (RAS)**

- Call pushes return address into the RAS
- When a return instruction decoded, pop the predicted return address from RAS

![Diagram showing the process of returning from a function using the Return Address Stack (RAS).]
**Solution: Return Address Stack (RAS)**

- Accurate prediction even with **small RAS**
- **8 ~ 16** number of return stack entries is good enough!

![Graph showing misprediction rate vs. number of entries in the return stack for different applications and processors.](image)
Examples in Real μProcessor Architecture

- Highly speculative O3 Super-scalar
- Using a hybrid predictor that combines local & global history components with a meta-predictor (selector)

- Hybrid predictor that combines local & global history predictors, but uses tag-based selection mechanism
- RAS/BTB is implemented, including a special component for indirect branches
  → Common for object-oriented code (vtables)
Trend for your studies?

- The trend of top-tier conferences: ISCA, ASPLOS, HPCA, MICRO
- Publication trends at ISCA
- First author affiliation trends over the last two decades of ISCA
Attacks related to Branch Target Prediction

Last lecture we covered...
Spectr1: Bound Check Bypass
Meltdown: Rogue Data Cache Load
**Spectre2: Branch Target Injection**

**CVE-2017-5715**

- **Goal of attack:** make victim process branch to attacker code and get the sensitive data of victim process

- **Key idea:**
  1. Attacker repeats to branch where the attacker code exist
  2. Attacker’s target is cached to BTB
  3. As BTB is sharable among all processes, victim process can branch to Attacker’s target
  4. Perform the cache channel attack!
Branch Prediction

Myoungsoo Jung
Computer Division

Computer Architecture and Memory systems Laboratory
Examples of Branch Correlations

- Example A: if the **first branch** is taken, always **next branch** will be taken
- Example B: if **branches 1 and 2** are taken, **branch 3** won’t be taken

→ Branch correlation based prediction could be done by keeping a history of the past “m” branches