20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming
February 7-11, 2015
Co-organized with:

CGO (International Symposium on Code Generation and Optimization)
HPCA (IEEE Symposium on High Performance Computer Architecture)
Brain-Inspired Computing

CSCI-564 Advanced Computer Architecture

Lecture 10: Handling Branches 2

Bo Wu
Colorado School of Mines
Last Time Predictor

• Last time predictor
  – Single bit per branch (stored in BTB)
  – Indicates which direction branch went last time it executed

  TTTTTTTTTTNNNNNNNNNNN → 90% accuracy

• Always mispredicts the last iteration and the first iteration of a loop branch
  – Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations
-- Loop branches for loops will small number of iterations

TNTNTNTNTNTNTNNTNTNTN → 0% accuracy

Last-time predictor CPI = [ 1 + (0.20*0.15) * 2 ] = 1.06  (Assuming 85% accuracy)
Implementing the Last-Time Predictor

The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch.
State Machine for Last-Time Prediction

- Predict not taken
- Predict taken
- Actually not taken
- Actually taken

Transition paths:
- From predict not taken to predict taken
- From predict taken to predict not taken
- From predict not taken to actually not taken
- From predict taken to actually taken
Improving the Last Time Predictor

• Problem: A last-time predictor changes its prediction from $T \rightarrow NT$ or $NT \rightarrow T$ too quickly
  – even though the branch may be mostly taken or mostly not taken

• Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  – Use two bits to track the history of predictions for a branch instead of a single bit
  – Can have 2 states for T or NT instead of 1 state for each

Two-Bit Counter (2BC) Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

Accuracy for a loop with N iterations = \((N-1)/N\)
TNTNTNTNTNTNTNTNTNTNTNT
→ 50% accuracy
(assuming counter initialized to weakly taken)

+ Better prediction accuracy
2BC predictor CPI = \([1 + (0.20*0.10) * 2]\) = 1.04 (90% accuracy)

-- More hardware cost
State Machine for 2-bit Saturating Counter

- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values

![State Machine Diagram]
Hysteresis Using a 2-bit Counter

Change prediction after 2 consecutive mistakes
Is This Enough?

• \(~85-90\%\) accuracy for many programs with 2-bit counter based prediction (also called bimodal prediction)

• Is this good enough?

• How big is the branch problem?
Rethinking the The Branch Problem

• Control flow instructions (branches) are frequent
  – 15-25% of all instructions

• Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  – N cycles: (minimum) branch resolution latency
  – Stalling on a branch wastes instruction processing bandwidth (i.e. reduces IPC)
    • N x W instruction slots are wasted (W: pipeline width)

• How do we keep the pipeline full after a branch?
• Problem: Need to determine the next fetch address when the branch is fetched (to avoid a pipeline bubble)
Importance of The Branch Problem

- Assume a 5-wide *superscalar* pipeline with 20-cycle branch resolution latency

- How long does it take to fetch 500 instructions?
  - Assume 1 out of 5 instructions is a branch
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = 120 cycles
    - 20% extra instructions fetched
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
    - 40% extra instructions fetched
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
    - 100% extra instructions fetched
Can We Do Better?

• Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  – Global branch correlation

• Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
  – Local branch correlation
Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

- If first branch not taken, second also not taken
  
  ```
  if (cond1)
  ...
  if (cond1 AND cond2)
  ```

- If first branch taken, second definitely not taken
  
  ```
  branch Y: if (cond1) a = 2;
  ...
  branch X: if (a == 0)
  ```
Global Branch Correlation (II)

branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)

• If Y and Z both taken, then X also taken
• If Y or Z not taken, then X also not taken
Global Branch Correlation (III)

- Eqntott, SPEC 1992

```c
if (aa==2)          ;; B1
    aa=0;
if (bb==2)          ;; B2
    bb=0;
if (aa!=bb) {
    ....
}                ;; B3
```

If B1 is not taken (i.e., aa==0@B3) and B2 is not taken (i.e. bb=0@B3) then B3 is certainly taken.
Capturing Global Branch Correlation

• Idea: Associate branch outcomes with "global T/NT history" of all branches

• Make a prediction based on the outcome of the branch the last time the same global branch history was encountered

• Implementation:
  – Keep track of the "global T/NT history" of all branches in a register → Global History Register (GHR)
  – Use GHR to index into a table that recorded the outcome that was seen for each GHR value in the recent past → Pattern History Table

• Global history/branch predictor

• Uses two levels of history (GHR + history at that GHR)
Two Level Global Branch Prediction

- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

How Does the Global Predictor Work?

for (i=0; i<100; i++)
    for (j=0; j<3; j++)

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

<table>
<thead>
<tr>
<th>test</th>
<th>result</th>
<th>value</th>
<th>GR</th>
</tr>
</thead>
<tbody>
<tr>
<td>j&lt;3</td>
<td>taken</td>
<td>j=1</td>
<td>1101</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>taken</td>
<td>j=2</td>
<td>1011</td>
</tr>
<tr>
<td>j&lt;3</td>
<td>not taken</td>
<td>j=3</td>
<td>0111</td>
</tr>
<tr>
<td>i&lt;100</td>
<td>usually taken</td>
<td>1110</td>
<td></td>
</tr>
</tbody>
</table>

Intel Pentium Pro Branch Predictor

• 4-bit global history register
• Multiple pattern history tables (of 2 bit counters)
  – Which pattern history table to use is determined by lower order bits of the branch address
Improving Global Predictor Accuracy

• Idea: Add more context information to the global predictor to take into account which branch is being predicted
  – Gshare predictor: GHR hashed with the Branch PC
  + More context information
  + Better utilization of PHT
  -- Increases access latency

Review: One-Level Branch Predictor

Direction predictor (2-bit counters)

Program Counter

Address of the current instruction

Cache of Target Addresses (BTB: Branch Target Buffer)

PC + inst size

Next Fetch Address

taken?

hit?
Two-Level Global History Branch Predictor

- Which direction earlier branches went
- Global branch history
- Program Counter
- Address of the current instruction

Direction predictor (2-bit counters)

PC + inst size

taken?

hit?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Two-Level Gshare Branch Predictor

- Global branch history
- Program Counter
- Which direction earlier branches went
- Address of the current instruction

Direction predictor (2-bit counters)

XOR

PC + inst size

Taken?

Hit?

Target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Can We Do Better?

• Last-time and 2BC predictors exploit only “last-time” predictability for a given branch

• Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  – Global branch correlation

• Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (in addition to the outcome of the branch “last-time” it was executed)
  – Local branch correlation
Local Branch Correlation

\begin{verbatim}
for (i=1; i<=4; i++) {}
\end{verbatim}

If the loop test is done at the end of the body, the corresponding branch will execute the pattern \((1110)^n\), where 1 and 0 represent taken and not taken respectively, and \(n\) is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

Capturing Local Branch Correlation

• Idea: Have a per-branch history register
  – Associate the predicted outcome of a branch with “T/NT history” of the same branch

• Make a prediction based on the outcome of the branch the last time the same local branch history was encountered

• Called the local history/branch predictor
• Uses two levels of history (Per-branch history register + history at that history register value)
Two Level Local Branch Prediction

- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

Pattern History Table (PHT)

Two-Level Local History Branch Predictor

Which directions earlier instances of *this branch* went

Program Counter

Address of the current instruction

Direction predictor (2-bit counters)

taken?

PC + inst size

hit?

target address

Cache of Target Addresses (BTB: Branch Target Buffer)

Next Fetch Address
Hybrid Branch Predictors

• Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the “best” prediction
  – E.g., hybrid of 2-bit counters and global predictor

• Advantages:
  + Better accuracy: different predictors are better for different branches
  + Reduced warmup time (faster-warmup predictor used until the slower-warmup predictor warms up)

• Disadvantages:
  -- Need “meta-predictor” or “selector”
  -- Longer access latency

Alpha 21264 Tournament Predictor

- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch

Branch Prediction Accuracy (Example)

Figure 13: Combined Predictor Performance by Benchmark
Biased Branches

• **Observation:** Many branches are biased in one direction (e.g., 99% taken)

• **Problem:** These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing “interference” in branch prediction tables and history registers

• **Solution:** Detect such biased branches, and predict them with a simpler predictor

Predication (Predicated Execution)

- **Idea:** Compiler converts control dependence into data dependence $\rightarrow$ branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

(normal branch code)  (predicated code)

if (cond) {
  b = 0;
} else {
  b = 1;
}
x = b+1

```
if (cond) {
  b = 0;
} else {
  b = 1;
}
x = b+1
```
Predicated Execution (II)

- Predicated execution can be high performance and energy-efficient

Predicated Execution
Fetch  Decode  Rename  Schedule  RegisterRead  Execute

Branch Prediction
Fetch  Decode  Rename  Schedule  RegisterRead  Execute

Pipeline flush!!
Predicated Execution (III)

- **Advantages:**
  - + Eliminates mispredictions for hard-to-predict branches

- **Disadvantages:**
  - -- Causes useless work for branches that are easy to predict
  - -- Additional hardware and ISA support
  - -- Cannot eliminate all hard to predict branches
    - -- Loop branches
Conditional Execution in the ARM ISA

• Almost all ARM instructions can include an optional condition code.

• An instruction with a condition code is executed only if the condition code flags meet the specified condition.
Idealism

• Wouldn’t it be nice
  – If the branch is eliminated (predicated) only when it would actually be mispredicted
  – If the branch were predicted when it would actually be correctly predicted

• Wouldn’t it be nice
  – If predication did not require ISA support
Wish Branches

• The **compiler** generates code (with wish branches) that can be executed **either** as predicated code **or** non-predicated code (normal branch code)

• The **hardware decides** to execute predicated code or normal branch code at run-time based on the confidence of branch prediction

• **Easy to predict**: normal branch code

• **Hard to predict**: predicated code

Wish Jump/Join

```
A
  \_T_\_
  \_N_\_
  \_C_\_
    \_D_\_
    \_B_\_

p1 = (cond)
branch p1, TARGET

mov b, 1
jmp JOIN

TARGET:
mov b, 0

B
  \_A_\_
  \_A_\_
  \_B_\_
    \_C_\_
      \_D_\_

p1 = (cond)
(!p1) mov b,1
(p1) mov b,0

normal branch code
predicated code
```

High Confidence

Wish Jump/Join

```
A
  \_wish jump\_
  \_A_\_
  \_B_\_
    \_C_\_
      \_D_\_

p1=(cond)

wish_jump p1 TARGET

B
  \_nop\_
  \_wish join
  \_A_\_
  \_A_\_
  \_B_\_
    \_C_\_
      \_D_\_

(!p1) mov b,1

wish_jump (!p1) JOIN

C
  \_TARGET:\_
  \_TARGET:
  \_TARGET:
    \_TARGET:

(p1) mov b,0

D
JOIN:

wish jump/join code
```
Dual-Path Execution versus Predication

A

Hard to predict

C

B

D

E

F

Dual-path

path 1

C

D

E

F

path 2

C

D

E

F

Predicated Execution

path 1

C

CFM

B

CFM

path 2

D

E

F