Lecture - 6

Instruction-Level Parallelism
Dynamic Branch Prediction
and
High Performance Instruction Delivery

Dr. Soner Onder
CS 4431
Michigan Technological University
Dynamic Branch Prediction

- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems

- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior
Static Branch Prediction

- To reorder code around branches, need to predict branch statically when compiling.

- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC

- More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:
Dynamic Branch Prediction

- Performance = $f(\text{accuracy, cost of misprediction})$
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
A Simple Branch Predictor

- Accessed early in the pipeline using the branch instruction PC
- Updated using the actual outcome.

Branch PC

Prediction

Actual outcome

0  Not taken
1  Taken
A Simple Branch Predictor

- What happens when we see the sequence of branches:
  - T N T N T N T N T N
  - (T T T T T T T T N)*
  - N N N T N N N T

- What is the branch misprediction rate for each of the cases assuming the predictor is initialized to zero?
Dynamic Branch Prediction

- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on next time through code, when it predicts exit instead of looping
Two bit branch prediction

Branch PC

Branch history table
Branch prediction buffer

Values
- 00 Not taken
- 01 Not taken
- 10 Taken
- 11 Taken
Two-bit predictor state diagram

Diagram showing the state transitions for a two-bit predictor. The states include 'Predict taken 11', 'Predict taken 10', 'Predict not taken 01', and 'Predict not taken 00'. The transitions are marked as 'Taken' or 'Not taken'.
Two-bit predictor

- A branch must miss twice before the prediction is changed.
- It is a specialization of an n-bit saturating counter scheme.
- Next state table:

<table>
<thead>
<tr>
<th>NT</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
N-Bit saturating counter

- 0 - 2^n-1 possible values:
  - 0000
  - 0001
  - 0010
  - 0011
  - 0100
  - 0101
  - 0110
  - 0111
  - 1000
  - 1001
  - 1010
  - 1011
  - 1100
  - 1101
  - 1110
  - 1111

- Increment upon taken
- Decrement upon not taken

PREDICT Not Taken

PREDICT Taken
BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table

- 4096 entry table:
Spec89 Prediction Accuracy, infinite buffer

SPEC89 benchmarks:

- nasa7: 1% (4096 entries), 0% (unlimited)
- matrix300: 0% (4096 entries), 0% (unlimited)
- tomcatv: 1% (4096 entries), 0% (unlimited)
- doduc: 5% (4096 entries), 5% (unlimited)
- spice: 9% (4096 entries), 9% (unlimited)
- fppp: 9% (4096 entries), 9% (unlimited)
- gcc: 12% (4096 entries), 11% (unlimited)
- espresso: 5% (4096 entries), 5% (unlimited)
- eqntott: 18% (4096 entries), 18% (unlimited)
- li: 10% (4096 entries), 10% (unlimited)

Frequency of mispredictions
Correlating (Two-Level) Branch Predictors

Consider the sequence:
- if (aa == 2) aa=0;
- if (bb == 2) bb=0;
- if (aa != bb) { ... }

What can you say about the behavior of the last branch with respect to the prior two branches?

MIPS assembly:
- aa is in R1, bb is in R2
  DSUBI R3,R1,#2
  BNEZ R3,L1 ; aa!=2
  DADD R1,R0,R0

- L1:
  DSUBI R3,R2,#2
  BNEZ R3,L2 ; bb != 2
  DADD R2,R0,R0

- L2:
  DSUBI R3,R1,R2
  BEQZ R3,L3 ; aa==bb
Correlating Branch Predictors

How can we capture the behavior of last n branches and adjust the behavior of the current branch accordingly?

Answer:
Use an n bit shift register, and shift the behavior of each branch to this register as they become known.

How many possible values will our shift register have?
Imagine that many tables and select the table you want to use based on the value of the shift register.
Correlated Branch Prediction

- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper n-bit branch history table.

- In general, (m,n) predictor means record last m branches to select between 2m history tables, each with n-bit counters.
  - Thus, old 2-bit BHT is a (0,2) predictor.

- Global Branch History: m-bit shift register keeping T/NT status of last m branches.

- Each entry in table has m n-bit predictors.
Correlating Branches

(2,2) predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction

2 Bits of global history means we look at T/NT behavior of last two branches to determine the behavior of THIS branch.

The buffer can be implemented as a one dimensional array. How?

(m,n) predictor uses behavior of last m branches to choose from 2m predictor each being an n-bit predictor.
(m,n) predictor uses behavior of last m branches to choose from $2^m$ predictor each being an n-bit predictor. How many bits are there in a (0,2) branch predictor that has 4K entries selected by the branch address?

$2^0 \times 2 \times 4K = 8K$.

How many bits does the example predictor have?

$2^2 \times 2 \times 16 = 128$ bits
What can you say about the performance of a (2,2) branch predictor?

Why is this the case?

Which benchmarks show this behavior? Why?
Gshare Correlating predictor

Branch PC

Global history: 0 1 1 0 1 0

XOR

Branch history table

What is happening here?
Quick Homework:

Show the working of a gshare predictor that uses two bits of global history, and having 16 entries.

For this, enumerate the possible PC values (i.e., the portion of the PC that is used to index the table) and the global history values. Record the number of times each table entry is referenced. If you like, you may write a small C program for this purpose.

Are there collisions?

Is this harmful?
Hybrid predictors

- The basic idea is to use a META predictor to select among multiple predictors.
- Example:
  - Local predictors are better in some branches.
  - Global predictors are better in utilizing correlation.
  - Use a predictor to select the better predictor.
Tournament Predictors

n/m means:
- n left predictor
- m right predictor

0: Incorrect
1: Correct

A predictor must be twice incorrect before we switch to the other one.
Tournament Predictors

- Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:
  - Global predictor
    - 4K entries index by history of last 12 branches \((2^{12} = 4K)\)
    - Each entry is a standard 2-bit predictor
  - Local predictor
    - Local history table: 1024 10-bit entries recording last 10 branches, index by branch address
    - The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters
Alpha 21264 Branch Prediction Mechanism

Source: Microprocessor Report, 10/28/96
The Tournament predictor selects between a local 2-bit predictor and a 2-bit Gshare predictor. Each predictor has 1024 entries each 2 bits for a total of 64 K bits.
Misprediction rates

Conditional branch misprediction rate

Total predictor size

Local 2-bit predictors
Correlating predictors
Tournament predictors

© 2003 Elsevier Science (USA). All rights reserved.
Pentium 4 Misprediction Rate (per 1000 instructions)

≈6% misprediction rate per branch SPECint
(19% of INT instructions are branch)

≈2% misprediction rate per branch SPECfp
(5% of FP instructions are branch)
Dynamic Branch Prediction Summary

- Prediction becoming important part of execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
  - Either different branches (GA)
  - Or different executions of same branches (PA)

- Tournament predictors take insight to next level, by using multiple predictors
  - Usually one based on global information and one based on local information, and combining them with a selector
  - In 2006, tournament predictors using $\approx 30K$ bits are in processors like the Power5 and Pentium 4

- Branch Target Buffer: include branch address & prediction, discuss next
Branch Target Buffers

- We need to know the branch target address as well as the direction of the branch.

- We need to supply the branch target *before* decoding the current instruction!

- Don’t worry there is a simple way to achieve this. It is called a BTB.
Branch Target Buffers

PC of instruction to fetch

Look up

Predicted PC

Number of entries in branch-target buffer

If instruction is not predicted to be a branch, proceed normally.

Yes: then instruction is branch and predicted PC should be used as the next PC.

No: branch predicted taken or not taken.
Branch Target Buffer - Steps

1. **Send PC to memory and branch-target buffer**

2. **Entry found in branch-target buffer?**
   - **No**
     - Normal instruction execution
   - **Yes**
     - **Send out predicted PC**

3. **Is instruction a taken branch?**
   - **No**
     - Normal instruction execution
   - **Yes**
     - **Taken branch?**
       - **No**
         - Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
       - **Yes**
         - Branch correctly predicted; continue execution with no stalls

4. **Enter branch instruction address and next PC into branch-target buffer**
Return Address Predictors

Procedure foo()
{
    Important stuff
    return;  {It really is jr $31}
}

for i=1; i < 100000; i++
{
    foo();
}

What can you say about the prediction accuracy of BTB for the jr instructions?
Procedure foo()
{
  Important stuff
  return;  {It really is jr $31}
}

for i=1; i < 50000; i++)
{
  foo();
  foo();
  foo();
}
Return Address Predictors

• Use a stack:
  • call (i.e. jal to a subroutine) push the return address onto the stack.
  • Return (i.e. jr $31) pop the address from the stack.
  • Discard the bottom entry if overflow.

What can you say about the prediction accuracy of BTB for the jr instructions if we have an infinite stack depth?

How about a limited stack depth?
Return Address Predictors

Misprediction rate

Number of entries in the return stack

- gcc
- fpppp
- espresso
- doduc
- li
- tomcatv
Instruction predication

- Avoid branch prediction by turning branches into conditionally executed instructions:
  - if (x) then A = B op C else NOP
  - If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
  - IA-64: 64 1-bit condition fields selected so conditional execution of any instruction
Conditional Move Instructions

Example

if x < y then
  a=a + 1
else
  a=a * 2

Code Sequence

lw  r11,x
lw  r12,y
slt r3,r11,r12
lw  r7,a
addi r8,1
sll r9,r7,1
cmov r9,r8,r3
sw  r9,a
Full predication

Example

\[
\begin{align*}
\text{if } x < y \text{ then} & \quad a = a + 1 \\
\text{else} & \quad a = a * 2
\end{align*}
\]

\[
p = x < y; \\
p: a = a + 1; \\
!p: a = a * 2;
\]

Code sequence:

\[
\begin{align*}
lw & \ T: r11, x \\
lw & \ T: r12, y \\
slt & \ T: r3, r11, r12 \\
lw & \ T : r7, a \\
addi & \ r3: r8, r7, 1 \\
sll & \ r3: r8, r7, 1 \\
sw & \ T : r9, a
\end{align*}
\]
Instruction predication

- Drawbacks to conditional instructions
  - Still takes a clock even if “annulled”
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline
Dynamic Branch Prediction Summary

- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
- Branch Target Buffer: include branch address & prediction
- Return address predictor: Works well for most procedure calls.
- Predicated Execution can reduce number of branches as well as number of mispredicted branches.