CS 6290
Branch Prediction
Control Dependencies

• Branches are very frequent
  – Approx. 20% of all instructions

• Can not wait until we know where it goes
  – Long pipelines
    • Branch outcome known after B cycles
    • No scheduling past the branch until outcome known
  – Superscalars (e.g., 4-way)
    • Branch every cycle or so!
    • One cycle of work, then bubbles for ~B cycles?
Surviving Branches: Prediction

- Predict Branches
  - And predict them well!

- Fetch, decode, etc. on the predicted path
  - Option 1: No execute until branch resolved
  - Option 2: Execute anyway (speculation)

- Recover from mispredictions
  - Restart fetch from correct path
Branch Prediction

- Need to know two things
  - Whether the branch is taken or not (direction)
  - The target address if it is taken (target)

- Direct jumps, Function calls
  - Direction known (always taken), target easy to compute

- Conditional Branches (typically PC-relative)
  - Direction difficult to predict, target easy to compute

- Indirect jumps, function returns
  - Direction known (always taken), target difficult
Branch Prediction: Direction

• Needed for conditional branches
  – Most branches are of this type
• Many, many kinds of predictors for this
  – Static: fixed rule, or compiler annotation
    (e.g. “BEQL” is “branch if equal likely”)
  – Dynamic: hardware prediction
• Dynamic prediction usually history-based
  – Example: predict direction is the same as the last time this branch was executed
Static Prediction

• Always predict NT
  – easy to implement
  – 30-40% accuracy ... not so good

• Always predict T
  – 60-70% accuracy

• BTFNT
  – loops usually have a few iterations, so this is like always predicting that the loop is taken
  – don’t know target until decode
One-Bit Branch Predictor

K bits of branch instruction address

Index

Branch history table of $2^K$ entries, 1 bit per entry

Use this entry to predict this branch:

0: predict not taken
1: predict taken

When branch direction resolved, go back into the table and update entry: 0 if not taken, 1 if taken
One-Bit Branch Predictor (cont’d)

0xDC08: for(i=0; i < 100000; i++)
{  
0xDC44: if( ( i % 100) == 0 )
    tick( );
0xDC50: if( (i & 1) == 1)
    odd( );
}
The Bit Is Not Enough!

• Example: short loop (8 iterations)
  – Taken 7 times, then not taken once
  – Not-taken mispredicted (was taken previously)

• Execute the same loop again
  – First always mispredicted
    (previous outcome was not taken)
  – Then 6 predicted correctly
  – Then last one mispredicted again

• Each fluke/anomaly in a stable pattern results in two mispredicts per loop
Examples

DC08: TTTTTTTTTTT ... TTTTTTTTNTTTTTTTT ... 100,000 iterations

How often is branch outcome != previous outcome?
2 / 100,000

DC44: TTTTT ... TTTTTTT ... TTTTTTT ... 2 / 100

99.998% Prediction Rate

DC50: TNTNTN TNTNTNTN TNTNTNTN TNTNTNTN ... 2 / 2

98.0%

NT

TN

0.0%
Two Bits are Better Than One

Predict NT
Predict T
Transition on T outcome
Transition on NT outcome

FSM for Last-Outcome Prediction

FSM for 2bC (2-bit Counter)
Example

Only 1 Mispredict per N branches now!
DC08: 99.999%  DC04: 99.0%
Still Not Good Enough

SPEC89 benchmarks

- nasa7: 1%
- matrix300: 0%
- tomcatv: 1%
- doduc: 5%
- spice: 9%
- fppp: 9%
- gcc: 12%
- espresso: 5%
- eqntott: 18%
- li: 10%

These are good

We can live with these

This is bad!
Importance of Branches

• 98% → 99%
  - Who cares?
  - Actually, it’s 2% misprediction rate → 1%
  - That’s a halving of the number of mispredictions

• So what?
  - If misp rate equals 50%, and 1 in 5 insts is a branch, then number of useful instructions that we can fetch is:
    \[ 5 \times (1 + \frac{1}{2} + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^3 + \ldots ) = 10 \]
  - If we halve the miss rate down to 25%:
    \[ 5 \times (1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \left(\frac{3}{4}\right)^3 + \ldots ) = 20 \]
  - Halving the miss rate doubles the number of useful instructions that we can try to extract ILP from
How about the Branch at 0xdc50?

• 1bc and 2bc don’t do too well (50% at best)
• But it’s still obviously predictable
• Why?
  – It has a repeating pattern: (NT) *
  – How about other patterns? (TTNTN) *

• Use branch correlation
  – The outcome of a branch is often related to previous outcome(s)
Idea: Track the History of a Branch

prev = 1  |  prev = 0  |  prediction
prev = 0  |  prev = 1  |  prediction
prev = 1  |  prev = 1  |  prediction
Deeper History Covers More Patterns

• What pattern has this branch predictor entry learned?

001 → 1; 011 → 0; 110 → 0; 100 → 1

00110011001... (0011)*
Global vs. Local Branch History

- **Local Behavior**
  - What is the predicted direction of Branch A given the outcomes of previous instances of Branch A?

- **Global Behavior**
  - What is the predicted direction of Branch Z given the outcomes of all* previous branches A, B, ..., X and Y?

* number of previous branches tracked limited by the history length
Why Global Correlations Exist

- Example: related branch conditions

A: \( p = \text{findNode}(\text{foo}) \);
   
   if ( \( p \) is parent )
   do something;

   do other stuff;  /* may contain more branches */

B: if ( \( p \) is a child )
   do something else;

Outcome of second branch is always opposite of the first branch
Other Global Correlations

• Testing same/similar conditions
  – code might test for NULL before a function call, and the function might test for NULL again
  – in some cases it may be faster to recompute a condition rather than save a previous computation in memory and re-load it
  – partial correlations: one branch could test for cond₁, and another branch could test for cond₁ & cond₂ (if cond₁ is false, then the second branch can be predicted as false)
  – multiple correlations: one branch tests cond₁, a second tests cond₂, and a third tests cond₁ ⊕ cond₂ (which can always be predicted if the first two branches are known).
Tournament Predictors

• No predictor is clearly the best
  – Different branches exhibit different behaviors
    • Some “constant”, some global, some local

• Idea:
  Let’s have a predictor to predict which predictor will predict better 😊
Tournament Hybrid Predictors

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

<table>
<thead>
<tr>
<th>Pred₀</th>
<th>Pred₁</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>
Common Combinations

• Global history + Local history
• “easy” branches + global history
  – 2bC and gshare
• short history + long history

• Many types of behaviors, many combinations
Direction Predictor Accuracy

- Local 2-bit predictors
- Correlating predictors
- Tournament predictors

Conditional branch misprediction rate vs. Total predictor size

© 2003 Elsevier Science (USA). All rights reserved.
Target Address Prediction

• Branch Target Buffer
  - IF stage: need to know fetch addr every cycle
  - Need target address one cycle after fetching a branch
  - For some branches (e.g., indirect) target known only after EX stage, which is way too late
  - Even easily-computed branch targets need to wait until instruction decoded and direction predicted in ID stage (still at least one cycle too late)
  - So, we have a quick-and-dirty predictor for the target that only needs the address of the branch instruction
Branch Target Buffer

- BTB indexed by instruction address
- We don’t even know if it is a branch!
- If address matches a BTB entry, it is predicted to be a branch
- BTB entry tells whether it is taken (direction) and where it goes if taken
- BTB takes only the instruction address, so while we fetch one instruction in the IF stage we are predicting where to fetch the next one from

Direction prediction can be factored out into separate table
Branch Target Buffer

- PC of instruction to fetch
- Look up
- Predicted PC
- Number of entries in branch-target buffer
- No: instruction is not predicted to be branch; proceed normally
- Yes: then instruction is branch and predicted PC should be used as the next PC
- Branch predicted taken or untaken
Return Address Stack (RAS)

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

• But return address is actually easy to predict
  – It is the address after the last call instruction
    that we haven’t returned from yet
  – Hence the Return Address Stack
Return Address Stack (RAS)

- Call pushes return address into the RAS
- When a return instruction decoded, pop the predicted return address from RAS
- Accurate prediction even w/ small RAS
Example 1: Alpha 21264

- Hybrid predictor
  - combines local history and global history components with a meta-predictor
Example 2: Pentium-M

- Also hybrid, but uses tag-based selection mechanism
Pentium-M (cont’d)

• Local component also has support for loops
  – accurately predict branches of the form \((T^kN)^*\)
Pentium-M (cont’d)

• Special target prediction for indirect branches
  – common in object-oriented code (vtables)
  – assumes correlation with global history