3 main concepts in computer architecture: locality (ref, exec); prediction & speculation; parallelism)

Pretty simple problem

Cleverness in the solutions & what they're based on

The nub of the problem:

- In what pipeline stage does the processor fetch the next instruction? **ifetch**
- If that instruction is a conditional branch, when does the processor know whether the conditional branch is taken (execute code at the target address) or not taken (execute the sequential code)? **exec**
- What is the difference in cycles between them? **2**

The cost of stalling until you know whether to branch

- number of cycles in between * branch frequency = the contribution to CPI due to branches

Predict the branch outcome to avoid stalling

Branch Prediction

Branch prediction:

- Resolve a branch hazard by predicting which path will be taken
- Proceed under that assumption
- Flush the wrong-path instructions from the pipeline & fetch the right path if wrong

Performance improvement depends on:

- whether the prediction is correct
  (producing correct predictions is most of the innovation)
- how soon you can check the prediction

Branch Prediction

Prediction, instruction scheduling

***Dynamic* branch prediction:**
- the prediction changes as program behavior changes
- branch prediction implemented in hardware for a runtime check
- common algorithm based on branch history
  - predict the branch **taken** if branched the last time
  - predict the branch **not-taken** if didn't branch the last time

Alternative: **static branch prediction**
- compiler-determined prediction
- fixed for the life of the program
- A likely algorithm?

Forward NT, backward T

Why work?

just like architects, compiler writes are designing for common case

Diff schemes, simple to complex

Branch Prediction Buffer

Branch prediction buffer
- small memory indexed by the lower bits of the address of a branch instruction during the fetch stage
- contains a 1-bit prediction
  (which path the last branch to index to this BPB location took)
- do what the prediction says to do
- if the prediction is **taken** & it is **correct**
  - only incur a one-cycle penalty **target addr calculation requires read reg**
- if the prediction is **not taken** & it is **correct**
  - incur no penalty – why?
- if the prediction is **incorrect**
  - change the prediction
  - also flush the pipeline – why?
  - penalty is the same as if there were no branch prediction – why?

perf:80% correct

if 10 bits, how many entries in BPB?

Ramif of just low bits? multiple br to same entry
Why not high? worse distribution in BPB

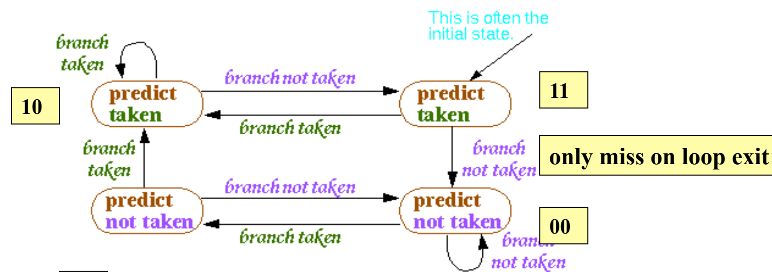
Two-bit Prediction

A single prediction bit does not work well with loops
- mispredicts the first & last iterations of a nested loop

common program construct

picture: mispredict on entry & exit

Two-bit branch prediction for loops
- Algorithm: have to be wrong twice in a row before prediction is changed



10

11

only miss on loop exit

00

01

+1,0,-1,-2
01 00 11 10
2 bits: prediction/strength

Sums up

Two-bit Prediction

Works well when branches predominantly go in one direction
- Why? A second check is made to make sure that a short & temporary change of direction does not change the prediction away from the dominant direction

What pattern is bad for two-bit branch prediction?

Design principle?
Common case

3bit not help more

perf:85

NT NT T T NT NT T T: pathological
satisfy the condition for changing & change back

What if going to design branch prediction today?

Is Branch Prediction More Important Today?

Think about:
- Is the number of branches in code changing?
- Is modern hardware design changing the dynamic frequency of branches?
- Is it getting harder to predict branch outcomes?
- Is the misprediction penalty changing?

Branch Prediction is More Important Today

Conditional branches still comprise about 20% of instructions

Correct predictions are more important today – why?
- **pipelines deeper**
  branch not resolved until more cycles from fetching
  therefore the **misprediction penalty** greater
  - cycle times smaller: more emphasis on throughput (performance)
  - more functionality between fetch & execute
- **multiple instruction issue** (superscalars & VLIW) & **multiple threads**
  branch occurs almost every cycle
  - flushing & re-fetching more instructions
- **object-oriented programming**
  more indirect branches which harder to predict

  Target is reg value
- **dual of Amdahl's Law**
  other forms of pipeline stalling are being addressed so the portion of CPI due to branch delays is relatively larger

All this means that the potential stalling due to branches is greater

Branch Prediction is More Important Today

On the other hand,

- chips are denser so we can consider sophisticated HW solutions
- hardware cost is small compared to the performance gain

Technical Directions in Branch Prediction

1: Improve the prediction

- correlated (2-level) predictor (Pentiums)
- use both history & branch address (MIPS, Sun)
- hybrid local/global predictor (Pentium 4, Power5)

2: Determine the target earlier

- branch target buffer (Pentium, Itanium)
- next address in I-cache (UltraSPARC)
- return address stack (everybody)

2 bits every cache block

3: Reduce misprediction penalty

- fetch both instruction streams (IBM mainframes)

4: Eliminate branch execution

- predicated execution (Itanium)

Either going to talk about the schemes or obvious what they do

1: Correlated Predictor

based on the short history of a single branch
but some branches are related, so maybe could do better

The rationale:

- having the prediction depend on the outcome of only 1 branch might produce bad predictions
- some branch outcomes are correlated

example: same condition variable

```
if (d==0)
...
if (d!=0)
```

example: related condition variable

```
if (d==0)
   b=1;
if (b==1)
```


1: Correlated Predictor

Can increase scope of analysis

more complicated example: related condition variables

```
if (x==2)                /* branch 1 */
   x=0;
if (y==2)                /* branch 2 */
   y=0;
if (x!=y)                /* branch 3 */
   do this; else do that;
```

- if branches 1 & 2 are taken, branch 3 is not taken

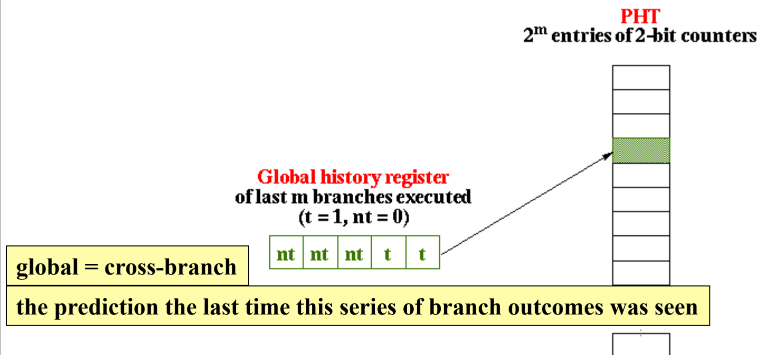
what this is telling us

⇒ use a **history of the past m branches**
represents an execution path through the program
(but still n bits of prediction)

1: Correlated Predictor

General idea of correlated branch prediction:

- put the global branch history in a global history register
  - global history is a shift register: shift left in the new branch outcome
- use its value to access a pattern history table (PHT) of 2-bit saturating counters



That's the basic idea

1: Correlated Predictor

Many implementation variations

- number of branch history registers
  - 1 history register for all branches (global)
  - table of history registers, 1 for each branch (private)
  - table of history registers, each shared by several branches (shared)
- history length (size of history registers)
- number of PHTs
- how access PHT
- What is the trade-off?

Combine history & branch address: history for this group of branches

accuracy vs hardware cost
evaluate in project

Typical: 10 history bits, 4K to 16K PHT entries

1: Tournament Predictor

Combine branch predictors

- local, per-branch prediction, accessed by the low PC bits
- correlated prediction based on the last *m* branches, assessed by the global history register
- indicator of which had been the best predictor for this branch
  - 2-bit counter: increase for one, decrease for the other

Predictor of the predictors

Determine target earlier. How determine target now? 1 cycle penalty

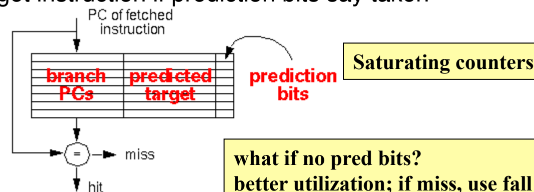
provides target in fetch stage **Small HW, lots of mileage**

Buffer (BTB)

Cache that stores: the addresses of branches
the predicted target address
branch prediction bits (optional)

Accessed by PC address in fetch stage
if hit: address was for *this* branch instruction
fetch the target instruction if prediction bits say taken

Why not just low bits?



Saturating counters

what if no pred bits?
better utilization; if miss, use fall thru code
argues for decoupling

No branch delay if: branch found in BTB
prediction is correct
(assume BTB update is done in the next cycles)

4K entries

BTB good for targets that don't change dynamically

2: Return Address Stack

The **bad** news:

- indirect jump targets are hard to predict
- registers for target calculation are accessed several stages after fetch

Proc call/proc ret/case

The **good** news: most indirect jumps (85%) are returns from functions

- optimize for this common case

Draw why BTB not work
Same br PC in func but diff target addr

Return address stack

- return address pushed on a call, popped on a return
- provides the return target early
- best for procedures that are called from multiple call sites
  - BTB would predict address of the return from the last call
- if "big enough", can predict returns perfectly
  - these days 1-32 entries

CAN SKIP

Reduce misprediction penalty

3: Fetch Both Targets

no or few caches
no or simple branch prediction

Fetch target & fall-through code

- reduces the misprediction penalty
- but requires lots of I-cache bandwidth
  - a dual-ported instruction cache ?
  - requires independent bank accessing ?
  - wide cache-to-pipeline buses

Why not used so much?
BW needed for wide I issue/PF/multiple threads
Ask yourself: what's the best use of this HW?

Eliminate br Is

4: Predicated Execution

Predicated instructions execute conditionally
- some other (previous) instruction sets a condition
- predicated instruction tests the condition & executes if the condition is true
- if the condition is false, predicated instruction isn't executed
- i.e., instruction execution is ***predicated*** on the condition

Eliminates conditional branch (expensive if mispredicted)
- changes a **control hazard** to a **data hazard**

what is data hazard

Fetch both true & false paths

4 Predicated Execution

Example:

without predicated execution

```
        sub   R10, R4, R5
        beqz  R10, Label
        add   R2, R1, R6
Label:  or    R3, R2, R7
```

0: or
1: add & or

with predicated execution

```
sub   R10, R4, R5
add   R2, R1, R6, R10
or    R3, R2, R7
```

generate side effects

Adv?
Disadv?

4 Predicated Execution

Is predicated execution a good idea?

Advantages of predicated execution

+ no branch hazard

  Might mispredict & penalty large if do

  especially good for hard to predict branches & deep pipelines
+ creates straightline code; therefore better prefetching of instructions

  prefetching = fetch instructions before you need them to hide instruction cache miss latency
+ more independent instructions, therefore better code scheduling

4 Predicated Execution

Disadvantages of predicated execution

- instructions on both paths are executed, structural hazard or more hardware if hardware is not idle

  Effect on perf/power
  - best for short code sequences
- hard to add predicated instructions to an existing instruction set
- additional register pressure

  Opcode/extra operand

  Explain reg alloc
- complex conditions if nested loops (predicated instructions may depend on multiple conditions)
- good branch prediction might get the same effect

IA-64
Pentium group negative data

Conditional move if only 1

Preview of calculating the cost of anything
What factors are involved?

Calculating the Cost of Branches

Factors to consider:

- branch frequency (every 4-6 instructions)
- correct prediction rate
  - 1 bit: ~ 80% to 85%
  - 2 bit: ~ high 80s to low 90%
  - correlated branch prediction: ~ 95%
- misprediction penalty
  RISCs: 4 -7 cycles
  Pentiums: larger, at least 9 cycles, 15 on average
  - then have to multiply by the instruction width
- or misfetch penalty
  have the correct prediction but not know the target address yet
  (may also apply to unconditional branches)

Calculating the Cost of Branches

What is the probability that a branch is taken?

Given:

- 20% of branches are unconditional branches
- of conditional branches,
  - 66% branch forward & are evenly split between taken & not taken
  - the rest branch backwards & are always taken

.2+.8(.66*.5+.33) approx = .7

Calculating the Cost of Branches

prediction is component metric
contribution to CPI is better execution bottom line

What is the contribution to CPI of conditional branch stalls, given:

- 15% branch frequency
- a BTB for conditional branches only with a
  - 10% miss rate
  - 3-cycle miss penalty
  - 92% prediction accuracy
  - 7 cycle misprediction penalty
- base CPI is 1

BTB result	Prediction	Frequency (per instruction)	Penalty (cycles)	*Stalls*
miss	--	.15 * .10 = .015	3	.045
hit	correct	.15 * .90 * .92 = .124	0	0
hit	incorrect	.15 * .90 * .08 = .011	7	.076
Total contribution to CPI				**.121**

Why no stalls? Fetch stage

Dynamic Branch Prediction, in Summary

Stepping back & looking forward,

how do you figure out whether branch prediction (or any other aspect of a processor) is still important to improve?

- Look at technology trends
- How do the trends affect different aspects of prediction performance (or hardware cost or power consumption, etc.)?
- Given these affects, which factors become bottlenecks?
- What techniques can we devise to eliminate the bottlenecks?

apply to dynamic branch prediction, as a way to summarize the discussion:
what causes a branch problem?
will it be a problem in the next 5-10 years?
what approaches do we take to find a solution?
what are particular solutions?

Prediction Research

- Predicting load addresses
- Predicting variable values
- Predicting which thread will hold a lock next
- Predicting which thread should execute on a multithreaded processor
- Predicting power consumption & when we can power-down processor components
- Predicting when a fault might occur **checkpoint if yes**