Pipeline Front-end
Instruction Fetch & Branch Prediction

Instructor: Nima Honarmand
Big Picture
Fetch Rate is an ILP Upper Bound

- Instruction fetch limits performance
  - To sustain IPC of N, must sustain a fetch rate of N per cycle
  - Need to fetch N on average, not on every cycle

- N-wide superscalar *ideally* fetches N instructions per cycle

- This doesn’t happen in practice due to:
  - Instruction cache organization
  - Branches
  - ... and interaction between the two
Instruction Cache Organization

- To fetch N instructions per cycle...
  - I$ line must be wide enough for N instructions

- PC register selects I$ line

- A *fetch group* is the set of insns. starting at PC
  - For N-wide machine, [PC,PC+N-1]
Fetch Misalignment (1/2)

- If PC = xxx01001, N=4:
  - Ideal fetch group is xxx01001 through xxx01100 (inclusive)

Misalignment reduces fetch width
Fetch Misalignment (2/2)

- Fetch block A and A+1 in parallel
  - Banked I$ + rotator network
    - To put instructions back in correct order
  - May add latency (add pipeline stages to avoid slowing down clock)

- There are other solutions using advanced data-array SRAM design techniques...

![Diagram showing banked I$ and rotator network](image)
Program Control Flow and Branches

• Program control flow is dynamic traversal of static CFG

• CFG is mapped to linear memory
Types of Branches

• **Direction:**
  – Conditional
    • Conditional branches
    • Can use Condition code (CC) register or General purpose register
  – Unconditional
    • Jumps, subroutine calls, returns

• **Target:**
  – PC-encoded
    • PC-relative
    • Absolute addr
  – Computed (target derived from register or stack)

Need direction and target to find next fetch group
What’s Bad About Branches?

1. Cause fragmentation of I$ lines

2. Cause disruption of sequential control flow
   – Need to determine \textit{direction}
   – Need to determine \textit{target}
Branches Disrupt Sequential Control Flow

- Need to determine target
  → Target prediction

- Need to determine direction
  → Direction prediction
Branch Prediction

• Why?
  – To avoid stalls in fetch stage (due to both unknown direction and target)

• Static prediction
  – Always predict not-taken (pipelines do this naturally)
  – Based on branch offset (predict backward branch taken)
  – Use compiler hints
  – These are all direction prediction, what about target?

• Dynamic prediction
  – Uses special hardware (our focus)
Dynamic Branch Prediction

• A form of speculation
  – Integrated with **Fetch** stage

• Involves three mechanisms:
  – Prediction
  – Validation (and training of the predictors)
  – Misprediction recovery

• **Prediction** uses two hardware predictors
  – **Direction predictor** guesses if branch is taken or not-taken
    • Applies to conditional branches only
  – **Target predictor** guesses the destination PC
    • Applies to all control transfers
BP in Superscalars

• Fetch group might contain multiple branches

• How many branches to predict?
  – Simple: upto the first one (now)
  – A bit harder: upto the first taken one (maybe later)
  – Even harder: multiple taken branches (maybe later)
    • Only useful if you can fetch multiple fetch groups from I$ in each cycle

• How to identify the branch and its target in Fetch stage?
  – I.e., without executing or decoding?
Option 1: Partial Decoding

- Huge latency (reduces clock frequency)
Option 2: Predeccoding

Predecode branches on fill from L2

Branch’s PC

L1-I

Pre decode branches on fill from L2

Store 1 bit per inst, set if inst is a branch

partial-decode logic removed

High latency (L1-I on the critical path)

sizeof(inst)

Target Pred

Dir Pred
Option 3: Using Fetch group Addr

• With one branch in fetch group, does it matter where it is?

• Fetch-group addr is stable
  – i.e., the same set of instructions are likely to be fetched using the same fetch group in the future
  – Why?

Latency determined by branch predictor
Target Prediction
Target Prediction

• Target: 32- or 64-bit value

• Turns out targets are generally easier to predict
  – Don’t need to predict not-taken target
  – Taken target doesn’t usually change

• Only need to predict taken-branch targets

• Predictor is really just a “cache”
  – Called \textit{Branch Target Buffer (BTB)}
Branch Target Buffer (BTB)
Set-Associative BTB

Next PC

PC

V tag target V tag target • • • V tag target
Making BTBs Cheaper

• Branch prediction is permitted to be wrong
  – Processor must have ways to detect mispredictions
  – Correctness of execution is always preserved
  – Performance may be affected

Can tune BTB accuracy based on cost
BTB w/Partial Tags

Fewer bits to compare, but prediction may alias
If target too far or PC rolls over, will mispredict
BTB Miss?

• Dir-Pred says “taken”

• Target-Pred (BTB) misses
  – Could default to fall-through PC (as if Dir-Pred said N-t)
    • But we know that’s likely to be wrong!

• Stall fetch until target known ... when’s that?
  – PC-relative: after decode, we can compute target
  – Indirect: must wait until register read/exec
Subroutine Calls

P: 0x1000: (start of printf)

A: 0xFC34: CALL printf

B: 0xFD08: CALL printf

C: 0xFFB0: CALL printf

BTB can easily predict target of calls
Subroutine Returns

P: 0x1000: ST $RA \rightarrow [\$sp]

0x1B98: LD $tmp \leftarrow [\$sp]
0x1B9C: RETN $tmp

A: 0xFC34: CALL printf
A’:0xFC38: CMP $ret, 0

B: 0xFD08: CALL printf
B’:0xFD0C: CMP $ret, 0

BTB can’t predict return for multiple call sites
Return Address Stack (RAS)

A: 0xFC34: CALL printf
   FC38
P: 0x1000: ST $RA \rightarrow [\$sp]
   …
   0x1B9C: RETN $tmp
A':0xFC38: CMP $ret, 0

Keep track of call stack
Return Address Stack Overflow

1. Wrap-around and overwrite
   - Will lead to eventual misprediction after four pops

2. Do not modify RAS
   - Will lead to misprediction on next pop
   - Need to keep track of # of calls that were not pushed

64AC: CALL printf

64B0

FC90
421C
48C8
7300

top of stack
Direction Prediction
Branches Have Locality

• If a branch was previously taken...
  – There’s a good chance it’ll be taken again

for(i=0; i < 100000; i++)
{
    /* do stuff */
}

This branch will be taken 99,999 times in a row.
Simple Direction Predictor

• Always predict N-t
  – No fetch bubbles (always just fetch the next line)
  – Does horribly on loops

• Always predict T
  – Does pretty well on loops
  – What if you have if statements?

```c
p = calloc(num,sizeof(*p));
if (p == NULL)
    error_handler();
```

This branch is practically never taken
Last Outcome Predictor

• Do what you did last time

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

How often is branch outcome != previous outcome?

2 / 100,000

99.998% Prediction Rate

98.0%

0.0%
Saturating Two-Bit Counter

FSM for Last-Outcome Prediction

FSM for 2bC (2-bit Counter)
Example

1bC: Initial Training/Warm-up

2bC:

Only 1 Mispredict per N branches now!
DC08: 99.999%  DC04: 99.0%

2x reduction in misprediction rate
Typical Organization of 2bC Predictor

- Hash can simply be the $\log_2 n$ least significant bits of PC
  - Or, something more sophisticated

PC $\rightarrow$ Hash

Hash $\rightarrow$ n entries/counters

Table update

FSM Update Logic

Prediction $\rightarrow$ Actual outcome
Dealing with Toggling Branches

• Branch at 0xDC50 changes on every iteration
  – 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*
  – Branch outcome is often related to previous outcome(s)
Track the *History* of Branches

- **prev = 1**: prediction = T
- **prev = 0**: prediction = T
- **prev = 1**: prediction = T
- **prev = 1**: prediction = T
- **prev = 1**: prediction = T
- **prev = 0**: prediction = T
- **prev = 0**: prediction = T
- **prev = 1**: prediction = N
- **prev = 1**: prediction = N
- **prev = 1**: prediction = T
- **prev = 1**: prediction = T
Deeper History Covers More Patterns

- Counters learn “pattern” of prediction

\[
\begin{array}{cccccc}
0 & 0 & 1 & 1 & 3 & 1 \rightarrow 0 & 3 \\
1 & 0 & 3 & 2 & 0 & 2 \rightarrow 1
\end{array}
\]

001 → 1; 011 → 0; 110 → 0; 100 → 1
00110011001… (0011)*
Predictor Organizations

Different pattern for each branch PC

Shared set of patterns

Mix of both
Branch Predictor Example (1/2)

• 1024 counters ($2^{10}$)
  – 32 sets
    • 5-bit PC hash chooses a set
  – Each set has 32 counters
    • $32 \times 32 = 1024$
    • History length of 5 ($\log_2{32} = 5$)

• Branch collisions
  – 1000’s of branches collapsed into only 32 sets
Branch Predictor Example (2/2)

• 1024 counters ($2^{10}$)
  – 128 sets
  • 7-bit PC hash chooses a set
  – Each set has 8 counters
    • $128 \times 8 = 1024$
    • History length of 3 ($\log_2 8 = 3$)

• Limited Patterns/Correlation
  – Can now only handle history length of three
Two-Level Predictor Organization

- **Branch History Table (BHT)**
  - $2^a$ entries
  - $h$-bit history per entry

- **Pattern History Table (PHT)**
  - $2^b$ sets
  - $2^h$ counters per set

- **Total Size in bits**
  - $h \times 2^a + 2^{(b+h)} \times 2$

Each entry is a 2-bit counter.
Classes of Two-Level Predictors

• h = 0 or a = 0 (Degenerate Case)
  – Regular table of 2bC’s (b = \log_2\text{counters})

• a > 0, h > 0
  – “Local History” 2-level predictor
    – Predict branch from its own previous outcomes

• a = 0, h > 0
  – “Global History” 2-level predictor
    – Predict branch from previous outcomes of all branches
Why Global Correlations Exist

Example: related branch conditions

A: \[ p = \text{findNode}(\text{foo}); \]
   \[ \text{if ( p is parent )}
       \text{do something;}
   \]
   \[ \text{do other stuff; /* may contain more branches */}

B: \[ \text{if ( p is a child )}
       \text{do something else;}
   \]

Outcome of second branch is always opposite of the first branch.
A Global-History Predictor

Single global Branch History Register (BHR)

PC Hash

b

h
Combined Indexing (1/2)

• “gshare” predictor (S. McFarling)
Combined Indexing (2/2)

• Not all $2^h$ “states” are used
  – (TTNN)* uses $\frac{1}{4}$ of the states for a history length of 4
  – (TN)* uses two states regardless of history length

• Not all bits of the PC are uniformly distributed
Tradeoff Between \( b \) and \( h \)

- Assume fixed number of counters

- Larger \( h \) → Smaller \( b \)
  - Larger \( h \) → longer history
    - Able to capture more patterns
    - Longer warm-up/training time
  - Smaller \( b \) → more branches map to same set of counters
    - More interference

- Larger \( b \) → Smaller \( h \)
  - Just the opposite...
Pros and Cons of Long Branch Histories

• Long global history provides context
  – More potential sources of correlation

• Long history incurs costs
  – PHT cost increases exponentially: $O(2^h)$ counters
  – Training time increases, possibly decreasing accuracy
    • Why decrease accuracy?
Predictor Training Time

- Ex: prediction equals opposite for 2\textsuperscript{nd} most recent

- Hist Len = 2
- 4 states to train:
  - $NN \rightarrow T$
  - $NT \rightarrow T$
  - $TN \rightarrow N$
  - $TT \rightarrow N$

- Hist Len = 3
- 8 states to train:
  - $NNN \rightarrow T$
  - $NNT \rightarrow T$
  - $NTN \rightarrow N$
  - $NTT \rightarrow N$
  - $TNN \rightarrow T$
  - $TNT \rightarrow T$
  - $TTN \rightarrow N$
  - $TTT \rightarrow N$
Combining Predictors

• Some branches exhibit local history correlations
  – ex. loop branches

• Some branches exhibit global history correlations
  – “spaghetti logic”, ex. if-elsif-elsif-elsif-else branches

• Global and local correlation often exclusive
  – Global history hurts locally-correlated branches
  – Local history hurts globally-correlated branches

• E.g., Alpha 21264 used hybrid of Gshare & 2-bit saturating counters
Tournament Hybrid Predictors

- Pred_0
- Pred_1
- Meta-Predictor

Final Prediction

If meta-counter MSB = 0, use Pred_0 else use Pred_1

<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>
Overriding Branch Predictors (1/2)

• Use two branch predictors
  – 1\textsuperscript{st} one has single-cycle latency (fast, medium accuracy)
  – 2\textsuperscript{nd} one has multi-cycle latency, but more accurate
  – Second predictor can \textit{override} the 1\textsuperscript{st} prediction

• E.x., in PowerPC 604
  – BTB takes 1 cycle to generate the target
    • Small 64-entry table
    • 1\textsuperscript{st} predictor: Predict taken if hit
  – Direction-predictor takes 2 cycles
    • Large 512-entry table
    • 2\textsuperscript{nd} predictor

Get speed without full penalty of low accuracy
Overriding Branch Predictors (2/2)

If A != A', flush A, B and C
restart fetch with A'

If A=A' (both preds agree), done
Speculative Branch Update (1/3)

• Ideal branch predictor operation
  1. Given PC, predict branch outcome
  2. Given actual outcome, update/train predictor
  3. Repeat

• Actual branch predictor operation
  – Streams of predictions and updates proceed parallel

Can’t wait for update before making new prediction
Speculative Branch Update (2/3)

- BHR update cannot be delayed until commit
  - But outcome not known until commit

Predict: \[A \quad B \quad C \quad D \quad E \quad F \quad G\]

Update: \[A \quad B \quad C \quad D \quad E \quad F \quad G\]

BHR:

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

Branches B-E all predicted with the same stale BHR value
Speculative Branch Update (3/3)

• Update branch history using predictions
  – *Speculative* update

• If predictions are correct, then BHR is correct

• What happens on a misprediction?
  – Can recover as soon as branch is resolved (EX)
  – Or, at retire stage
  – More details in recovery slides
Validation, Training & Misprediction Recovery
Validating Branch Outcome (1/2)

• Need to validate both target and prediction
  – Each one might be calculated at different stages of pipeline
    • Depending on the branch type
    • E.g., direction of unconditional branch is known in Decode stage
    • E.g., target of register-indirect-with-offset branch is known in Execute stage
  – Can validate each one separately
    • As soon as the correct answer is determined
  – Or, both at the same time
    • For example, after “executing” the branch in the execute stage
Validating Branch Outcome (1/2)

• Validation involves
  – Training of the predictors (always)
  – Misprediction recovery (if mispredicted)

• Training involves updating both predictors
  – Might need some extra information such as BHR used in prediction
  – Should keep this information somewhere to use for training

• Misprediction recovery involves
  – Re-steering fetch to correct address
  – Recovering correct pipeline state
    • Mainly squashing instructions from the wrong path
    • But also, other stuff like predictor states, RAS content, etc.
Misprediction Recovery

• Two options
  – Can wait until the branch reaches the head of ROB (slow)
    • And then use the same rewind mechanism as exceptions
  – Initiate recovery as soon as misprediction determined (fast)
    • requires checkpoint of all the state needed for recovery
    • should be able to handle out-of-order branch resolution

• Fast branch recovery
  – Invalidate all instructions in pipeline front-end
    • Fetch, Decode and Dispatch stage
  – Invalidate all insns in the pipeline back-end that depend on the branch
  – Use the checkpoints to recover data-structure states
Fast Branch Recovery

Key Ideas:

- For branches, keep copy of all state needed for recovery
  - Branch stack stores recovery state

- For all instructions, keep track of pending branches they depend on
  - Branch mask register tracks which stack entries are in use
  - Branch masks in RS/FU pipeline indicate all older pending branches
Fast Branch Recovery – Dispatch Stage

- **Branches:**
  - If branch stack is full, stall
  - Allocate stack entry, set **b-mask** bit
  - Take snapshot of map table, free list, ROB, LSQ tails
  - Save PC & details needed to fix BP

- **All instructions:**
  - Copy **b-mask** to RS entry
Fast Branch Recovery - Misprediction

- **Fix ROB & LSQ:**
  - Set tail pointer from branch stack

- **Fix Map Table & free list:**
  - Restore from checkpoint

- **Fix RS & FU pipeline entries:**
  - Squash if b-mask bit for branch == 1

- **Clear branch stack entry, b-mask bit**
  - Can handle nested mispredictions!
Fast Branch Recovery – Correct Prediction

- Free branch stack entry
- Clear bit in b-mask
- Flash-clear b-mask bit in RS & pipeline:
  - Frees b-mask bit for immediate reuse
- Branches may resolve out-of-order!
  - b-mask bits keep track of unresolved control dependencies