Pipeline Front-End
(Instruction Fetch & Branch Prediction)

Nima Honarmand
Big Picture

Instruction Flow

Register Data Flow

Memory Data Flow
Fetch Rate is an ILP Upper Bound

• Instruction fetch limits performance
  – To sustain IPC of N, must fetch N insts. per cycle
  – N on average, some cycles even more than N

• N-wide superscalar *ideally* fetches N insts. per cycle

• This doesn’t happen in practice due to:
  – Instruction cache organization
  – Branches
  – and the 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 instructions to be fetched
  – For N-wide machine, [PC, PC+N-1]
Problem: Fetch Misalignment

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

Misalignment reduces fetch width
Reducing Fetch Misalignment

- 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 the clock down)
Next Problem: Branches

Branch Classification:

• **Direction-wise:**
  – Conditional
    • Conditional branches
    • Can use Condition code (CC) register or General purpose register
  – Unconditional
    • Jump, subroutine call, return

• **Target-wise:**
  – Instruction-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 **direction** and **target** before fetching next fetch group
Branches Disrupt Sequential Control Flow

- It can take multiple cycles to calculate branch direction and target
- Naïve design would stall Fetch stage until that happens
- High-perf. designs use prediction for both
  - Direction prediction
  - Target prediction
- Two orthogonal issues!
Branch Prediction Types

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

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

- A form of speculation
  - Integrated with **Fetch** stage

Requires three mechanisms in hardware:
- **Prediction**
- **Validation** and **training** of the predictors
- **Misprediction recovery**

**Prediction** uses two hardware predictors
- **Direction predictor** guesses if branch is taken (just conditional branches)
- **Target predictor** guesses the destination PC (applied to all branches)
Target Prediction
Target Prediction

- Target: 32- or 64-bit instruction address
- Turns out targets are generally easier to predict
  - Taken target doesn’t usually change
- Only need to predict taken-branch targets
- Predictor is really just a “cache”
  - Called Branch Target Buffer (BTB)
Branch Target Buffer (BTB)

- Branch Instruction (Fetch Group) Address
- Valid Bit
- Hit?
- Next PC
- Branch Target Address
Set-Associative BTB

PC

\[ \text{tag} \rightarrow \text{target} \rightarrow \text{tag} \rightarrow \text{target} \rightarrow \cdots \rightarrow \text{tag} \rightarrow \text{target} \rightarrow \text{Next PC} \]
Making BTBs Cheaper

• Take advantage of the fact that 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
BTB w/PC-offset Encoding

If target too far from PC, will mispredict
BTB Miss?

• Suppose direction predictor says “taken”, and target predictor (BTB) misses

• Could default to fall-through PC (as if Dir-Pred said NT)
  – 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
BTB and Subroutine Calls

- BTB can easily predict target of most calls because they don’t change
- But some calls do change their targets
  - Example?
    - Virtual function calls in C++
    - BTB can still be effective if they don’t change too much
How about Subroutine Returns?

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

- Keep track of the call stack in a HW structure (RAS)
- When executing CALL, put return addr (i.e., inst after CALL) on top of RAS
- When executing RET, use address on top of RAS as target prediction

A: 0xFC34: CALL printf
   FC38
P: 0x1000: ST $RA → [$sp]
   ...
0x1B9C: RETN $tmp
A+4: 0xFC38: CMP $ret, 0
Return Address Stack Overflow

• What to do if RAS is full?
  – Can happen if call stack too deep

1) Wrap-around and overwrite
  • Will lead to eventual misprediction (after four pops in this example)

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

In practice, most processors use solution #1.
Direction Prediction
Branches Are Not Memory-Less

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

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

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

• Always predict N (not taken)
  – No fetch bubbles (always just fetch the next line)
  – Performs horribly on loops

• Always predict T
  – Performs pretty well on (long) loops
  – But, 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?

- 0xDC08: 
  - 100,000 iterations
  - 2 / 100,000

- 0xDC44:
  - 2 / 100

- 0xDC50:
  - 2 / 2

99.998% Prediction Rate

98.0%

0.0%
Saturating Two-Bit Counter

Predict N
Predict T
Transition on T outcome
Transition on N outcome

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 over 1bC
• Hash can simply be the $\log_2 n$ least significant bits of PC
  – Or, something more sophisticated
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)
Idea: Track the *History* of Branches

prev = 1  3  0  prediction = T ×
prev = 0  3  2  prediction = T
prev = 1  3  2  prediction = T
prev = 1  3  3  prediction = T
prev = 1  3  0  prediction = N
prev = 0  3  0  prediction = T
prev = 1  3  0  prediction = N
prev = 0  3  0  prediction = T
Deeper History Covers More Patterns

• Counters learn “pattern” of prediction

Branch outcomes: 001 1001 1001…  
Pattern: (0011)*  
001 → 1; 011 → 0; 110 → 0; 100 → 1
Predictor Organizations

- Limited counter budget → aliasing is inevitable
  - Different organizations trades off aliasing in different places

- PC Hash
  - Shared set of patterns

- PC Hash 1
  - Mix of both

- PC Hash 2
  - Different pattern for each branch PC
Branch Predictor Example (1)

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

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

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

- Limited Patterns/Correlation
  - Can now only handle history length of three
Two-Level Predictor Organization (1)

• In practice, keeping a separate history ($h$ bits) and a set of counters ($2^h$ counters) for each branch would waste too much space
  – Many branches, only have few valid histories, thus wasting counters corresponding to unused histories

• To reduce waste, we can use a two-level predictor organization consisting of two tables
  – **Branch History Table (BHT)**: tracks branch histories
  – **Pattern History Table (PHT)**: contains the $2bC$ counters
Two-Level Predictor Organization (2)

- **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

• $h = 0$ (Degenerate Case)
  – Regular table of $2bC'$s ($b = \log_2(#\text{counters})$)

• $a > 0$, $h > 0$
  – "Local History" two-level predictor
  – Predict branch from its own (and aliasing branches’) previous outcomes

• $a = 0$, $h > 0$
  – "Global History" two-level predictor
  – Predict branch from previous outcomes of all branches
  – Useful due to global branch correlations
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
A Global-History Predictor
gshare Global Predictor

- For a fixed number of counters, there is a trade-off between \( h \) (history length) and \( b \) (number of branches)

- Observation: in the previous design, 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

- “gshare” predictor (McFarling 1993) combines PC and global history for better counter utilization
Tradeoff Between $b$ and $h$

- Assume fixed number of counters

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

- Larger $b \rightarrow$ Smaller $h$
  - 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
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
Tournament Predictors (1)

- Some branches exhibit local history correlations
  - E.g., 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

- Idea: use hybrid designs consisting of both types of predictors
  - E.g., Alpha 21264 used hybrid of gshare (global) & simple table of 2bCs with no history (local)
Tournament Predictors (2)

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>

Final Prediction

Meta-Predictor

table of 2-bit counters
Overriding Branch Predictors

• Large (more accurate) predictors have higher latency
  – Either slow down the clock, or stall fetch for multiple cycles until predictor generates its result
  ✗ Both are bad options

• Idea: 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.g., 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)

- **Fast 1st Pred**
  - Predict A
  - Predict A'
  - Fetch A
  - Predict B
  - Predict B'
  - Fetch A

- **2-cycle Pipelined L1-I**
  - Predict C
  - Fetch B
  - Predict B'
  - Fetch A
  - Predict C'
  - Predict A'

If A = A' (both preds agree), done

If A ≠ A', flush A, B and C, restart fetch with A'
Speculative Branch Update (1)

• 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 in parallel

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

Update: \[ \text{A} \quad \text{B} \quad \text{C} \quad \text{D} \quad \text{E} \quad \text{F} \quad \text{G} \]
Speculative Branch Update (2)

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

Predict: A B C D E F G
Update: A B C D E F G

BHR: 0 0 0 0 0 0 0

Branches B-E all predicted with the same stale BHR value

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

• Update branch history using predictions
  – Speculative update

• If predictions are correct, then BHR is correct

• What happens on a misprediction?
  – Should recover as soon as branch is resolved (EX)
  – More details in recovery slides
Other Branch Prediction Approaches

• These BP styles are the foundation of many of modern BPs in use today
  – But there are many variations of these or other proposed techniques

• Examples:
  – Loop predictor: used in Intel processors
    • Predicts number of loop iterations to avoid end-of-loop misprediction
  – Perceptron predictor: rumored to be used in some Samsung & AMD processors
    • Uses a perceptron-like mechanism to assign weights to correlation of a given branch with previous branches to allow much larger histories
  – Tagged hybrid predictors: rumored to be used in recent Intel procs
    • Uses multiple predictors (each with a different history length) and a meta-predictor to select among them
Validation, Training & Misprediction Recovery
Validating Branch Outcome (1)

• Need to validate both target and direction
  – Each 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 (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 in pipeline registers 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
  1) Can wait until the branch reaches the head of ROB (slow)
     • And then use the same abort-and-restart mechanism as exceptions
  2) 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 back-end that depend on branch
    • Need a mechanism to identify branch-dependent instructions
  – Use **checkpoints** to recover data-structure states
Fast Branch Recovery

Key Ideas:

• On prediction, 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 entry indicate all older pending branches
Fast Branch Recovery – Dispatch Stage

• For branch instructions:
  – If branch stack is full, stall
  – Allocate stack entry, set \textbf{b-mask} bit
  – Take snapshot of map table, free list, ROB, LSQ tails, etc.
  – Save PC & details needed to fix Branch Predictors (BP)

• All instructions:
  – Copy \textbf{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

• This design 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 all unresolved control dependencies