Computer Architecture

Advanced Branch Prediction

By Yoav Etsion and Dan Tsafrir
Presentation based on slides by David Patterson, Avi Mendelson, Lihu Rappoport, and Adi Yoaz
Introduction

Given an instruction, need to predict if it’s a branch and…

- Branch type, namely determine if the branch is
  - Conditional / unconditional; direct / indirect; call / return / other

- For conditional branch, need to determine “direction”
  - Direction mean: “taken” or “not taken”
  - Actual direction is known only after execution
  - Wrong direction prediction => full pipeline flush

- For taken branch (cond. on uncond.), need to determine “target”
  - Target of direct branches known at decode
  - Target of indirect branches known at execution

Goal

- Minimize branch misprediction rate (for a given predictor size)
What/Who/When We Predict/Fix

**Fetch**
- **Target Array**
  - Branch type
    - conditional
    - unconditional direct
    - unconditional indirect
    - call
    - return
  - Branch target
- **Cond. Branch Predictor**
  - Predict conditional T/NT
- **Return Stack Buffer**
  - Predict return target
- **Indirect Target Array**
  - Predict indirect target
  - override TA target

**Decode**
- Fix TA miss
- Fix wrong (direct) target
- on TA miss try to fix

**Execute**
- Fix Wrong prediction
- Fix Wrong prediction
- Fix Wrong prediction

Dec Flush
Exe Flush
Branches and Performance

- MPI: misprediction-per-instruction:

\[
MPI = \frac{\text{# of incorrectly predicted branches}}{\text{total # of instructions}}
\]

- How is this different from misprediction rate?
  - The number of branch instructions in the code is highly workload-specific
  - MPI takes the rate of branches into account
Branches and Performance

- **MPI**: misprediction-per-instruction:
  \[ MPI = \frac{\text{# of incorrectly predicted branches}}{\text{total # of instructions}} \]

- **MPI correlates well with performance. For example:**
  - MPI = 1% (1 out of 100 instructions @ 1 out of 20 branches)
  - Avg. IPC=2; flush penalty of 10 cycles

- **We get:**
  - MPI = 1% ⇒ flush in every 100 instructions
  - Since IPC=2, we have 1 flush every 50 cycles
  - 10 cycles flush penalty every 50 cycles
  - 20% in performance
Branch Target Buffer (reminder)

- BTB is accessed using the branch address (branch IP)
- Implemented as an $n$-way set associative cache
  - Tags are usually partial, which saves space, but...
  - Can get false hits when a few branches are aliased to same entry
  - Luckily, it’s not a correctness issue (only performance)
- BTB predicts the following
  - Is the instruction a branch?
  - Target
  - Branch type
    - Unconditional $=>$ take target
    - Conditional $=>$ predict direction
- BTB maintenance
  - Allocated & updated at runtime, during execution

<table>
<thead>
<tr>
<th>Branch IP</th>
<th>tag</th>
<th>target</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

hit / miss (indicates a branch)  predicted target  predicted type
Predicting Direction of Conditional Branch:

“Taken” or “Not Taken”? 
One problem with 1-bit predictor:

- Double-mistake in loops

<table>
<thead>
<tr>
<th>Branch Outcome</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1</td>
<td>? 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1</td>
</tr>
</tbody>
</table>
Bimodal (2-bit) Predictor

- A 2-bit saturating counter avoids the double mistake in glitches
  - Need “more evidence” to change prediction
- 2 bits encode one of 4 states
  - 00 – strong NT, 01 – weakly NT, 10 – weakly taken, 11 – strong taken
  - Commonly initialized to “weakly-taken”

**Update**

- Branch was actually taken: increment counter (saturate at 11)
- Branch was actually not-taken: decrement counter (saturate at 00)

**Predict according to MSB of counter (0 = NT, 1 = taken)**
Bimodal Predictor (cont.)

array of 2-bit saturating counters

lsb-s of branch IP → Prediction = MSB of counter

Update counter with branch outcome
++ if taken
-- if not take (avoid overflowing)

Problem:
- Doesn’t predict well with patterns like 010101… (see example next slide)
Bimodal Predictor - example

- Br1 prediction
  - Pattern: \(1 \ 0 \ 1 \ 0 \ 1 \ 0\)
  - Counter: \(2 \ 3 \ 2 \ 3 \ 2 \ 3\)
  - Prediction: T T T T T T T

- Br2 prediction
  - Pattern: \(0 \ 1 \ 0 \ 1 \ 0 \ 1\)
  - Counter: \(2 \ 1 \ 2 \ 1 \ 2 \ 1\)
  - Prediction: T nT T nT T nT

- Br3 prediction
  - Pattern: \(1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0\)
  - Counter: \(2 \ 3 \ 3 \ 3 \ 3 \ 3 \ 3\)
  - Prediction: T T T T T T T

Code:
- Loop: ....
- br1: if (n/2) {
  /*odd*/ ........ }
- br2: if ((n+1)/2) {
  /*even*/ ........ }
- n--
- br3: JNZ n, Loop
2-level predictors

- More advanced branch predictors work in 2 levels

- There are local predictors
  - A branch $B$ can be predicted based on past behavior of $B$

- And global predictors
  - $B$ is mostly affected by nearby branches
Local Predictor

- Save the history of each branch in a Branch History Register (BHR):
  - Shift-register updated by branch outcome (new bit in => oldest bit out)
  - Saves the last $n$ outcomes of the branch
  - Used as a pointer to an array of bits specifying direction per history

- Example: assume $n=6$
  - Assume the pattern 000100010001 \ldots
  - At the steady-state, the following patterns are repeated in the BHR:

\[
\begin{align*}
000100010001 & \ldots \\
000100 & \\
010001 & \\
100010 & \\
\end{align*}
\]

- Following 000100, 010001, 100010 the jump is not taken
- Following 001000 the jump is taken
Local Predictor (2\textsuperscript{nd} level)

- Like before, there could be glitches from the pattern
  - Use 2-bit saturating counters instead of 1 bit to record outcome:
    - Too long BHRs are not good:
      - Distant past history may be no longer relevant
      - Warmup is longer
      - Counter array becomes too big ($2^n$)

![Diagram of 2-bit-sat counter array]
Local Predictor: private counter arrays

Holding BHRs and counter arrays for many branches:

Predictor size: \#BHRs \times (\text{tag\_size} + \text{history\_size} + 2 \times 2^{\text{history\_size}})

Example: \#BHRs = 1024; \text{tag\_size}=8; \text{history\_size}=6 \Rightarrow

size=1024 \times (8 + 6 + 2 \times 2^6) = 142\text{Kbit} (\text{too big})
Reducing size: shared counter arrays

- Using a single counter array shared by all BHR entries
  - All BHRs index the same array (2nd level is shared)
  - Branches with identical history interfere with each other (though, empirically, it still works reasonably well)

![Diagram of branch prediction system](image)

**Branch IP**

<table>
<thead>
<tr>
<th>tag</th>
<th>history1</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>history2</td>
</tr>
</tbody>
</table>

**2-bit-sat counter array**

**prediction** = msb of counter

Predictor size: \( \#\text{BHRs} \times (\text{tag}_\text{size} + \text{history}_\text{size}) + 2 \times 2^{\text{history}_\text{size}} \)

Example: \( \#\text{BHRs} = 1024; \text{tag}_\text{size}=8; \text{history}_\text{size}=6 \implies \text{size}=1024 \times (8 + 6) + 2 \times 2^6 = 14.1\text{Kbit} \text{ (much smaller)} \)
Local Predictor: *lselect*

- *lselect* reduces inter-branch-interference in the counter array by concatenating some IP bits to the BHRs, thereby making the counter array longer.

Predictor size: \( \#\text{BHRs} \times (\text{tag\_size} + \text{history\_size}) + 2 \times 2^{\text{history\_size} + m} \)

=> the 2bit array is \(2^m\) bigger (overall, a small addition for small \(m\))
Local Predictor: *Ishare*

*Ishare* reduces inter-branch-interference in the counter array with XOR: (maps common patterns of different branches to different counters)

Predictor size: \( \#BHRs \times (\text{tag\_size} + \text{history\_size}) + 2 \times 2 \) \( \text{history\_size} \)
Global Predictor

- Sometimes, a branch’s behavior tightly correlates with that of other branches:

  if \( x < 1 \) . . .
  if \( x > 1 \) . . .

- Using a Global History Register (GHR), the prediction of the second \texttt{if} may be based on the direction of the first \texttt{if}
  - Used for all conditional branches

- Yet, for other branches such history “interference” might be destructive
  - To compensate, need long history
Global Predictor (cont.)

The predictor size: \( \text{history}_\text{size} + 2 \times 2^{\text{history}_\text{size}} \)

Example: \( \text{history}_\text{size} = 12 \Rightarrow \text{size} = 8 \text{ K Bits} \)
gshare combines the global history information with the branch IP using XOR (again, maps common patterns of different branches to different counters)

Global Predictor: Gshare

2-bit-sat counter array

GHR

history

Branch IP

prediction = msb of counter

update counter with branch outcome

update history with branch outcome
Hybrid (Tournament) Predictor

A tournament predictor dynamically selects between 2 predictors:

Use the predictor with better prediction record (example: Alpha 21264)

- Note: the chooser array may also be indexed by the GHR

![Diagram of the tournament predictor]

- Chooser array (an array of 2-bit sat. counters)
- GHR
- Branch IP
- Global
- Bimodal / Local
- Prediction

++ if Bimodal / Local correct and Global wrong
-- if Bimodal / Local wrong and Global correct
Speculative History Updates

- Deep pipeline $\Rightarrow$ many cycles between fetch and branch resolution
  - If history is updated only at resolution
    - Local: future occurrences of the same branch may see stale history
    - Global: future occurrences of all branches may see stale history
  - History is speculatively updated according to the prediction
    - History must be corrected if the branch is mispredicted
    - Speculative updates are done in a special field to enable recovery

- Speculative History Update
  - Speculative history updated assuming previous predictions are correct
  - Speculation bit set to indicate that speculative history is used
  - As usual, counter array updated only when outcome is known (that is, it is not updated speculatively)

- On branch resolution
  - Update the real history (needed only for misprediction) and counters
“Return” Stack Buffer

- A return instruction is a special case of an indirect branch:
  - Each time jumps to a potentially different target
  - Target is determined by the location of the corresponding call instruction

- The idea:
  - Hold a small stack of targets
  - When the target array predicts a call
    - Push the address of the instruction which follows the call-instruction into the stack
  - When the target array predicts a return
    - Pop a target from the stack and use it as the return address
Branch Prediction in commercial Processors
Real World Predictors

- **386 / 486**
  - All branches are statically predicted “Not Taken”

- **Pentium**
  - IP based, 2-bit saturating counters (Lee-Smith)
    - An array indexed by part of IP bits
  - Upon predictor miss (IP not in found in array)
    - Statically predicted “not taken”
Intel Pentium III

- 2-level, local histories, per-set counters
- 4-way set associative: 512 entries in 128 sets

Return Stack Buffer

Branch Type
- 00 - cond
- 01 - ret
- 10 - call
- 11 - uncond

Pred = msb of counter
Alpha 21264 - LG Tournament

- New entry on the Local stage is allocated on a global stage miss-prediction
- Chooser state-machines: 2 bit each:
  - one bit saves last time global correct/wrong,
  - and the other bit saves for the local correct/wrong
- Chooses Local only if local was correct and global was wrong

In each entry:
- 6 bit tag + 10 bit History

GHR

Global

Local

Choose
Pentium® M

- Combines 3 predictors
  - Bimodal, Global and Loop predictor
- Loop predictor analyzes branches to see if they have loop behavior
  - Moving in one direction (taken or NT) a fixed number of times
  - Ended with a single movement in the opposite direction
## Pentium® M – Indirect Branch Predictor

- **Indirect branch targets is data dependent**
  - Can have many targets: e.g., a case statement
  - Can still have only a single target at run time
  - Resolved at execution ⇒ high misprediction penalty

- **Used in object-oriented code (C++, Java)**
  - becomes a growing source of branch mispredictions

- **A dedicated indirect branch target predictor (iBTB)**
  - Chooses targets based on a global history (similar to global predictor)

- **Initially indirect branch is allocated only in the BTB**
  - If target is mispredicted ⇒ allocate an iBTB entry corresponding to the
global history leading to this instance of the indirect branch
  - Data-dependent indirect branches allocate as many targets as needed
  - Monotonic indirect branches are still predicted by the TA
Indirect branch target prediction (cont)

- Prediction from the iBTB is used if
  - BTB indicates an indirect branch
  - iBTB hits for the current global history (XORed with branch address)

```
<table>
<thead>
<tr>
<th>Branch IP</th>
<th>BTB</th>
<th>iBTB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Global history
- Predicted Target
- HIT
- Hit
- Indirect branch
- Target
- MUX
- Predicted Target
Summary

- Branches are frequent
- Branches are bad (for performance)
- Branches are predictable...

- Speculating branch outcome improve pipeline utilization

- Speculation better be accurate:
  - Remember the example: a single mispredicted branch per 100 instructions can reduce performance by 20% (IPC=2)

- It is effective to spend a lot of transistors on branch predictors
  - Prediction accuracy is typically over 96%