CISC pitfalls and Branch Prediction
The CISC pitfalls and solutions

- Self Modifying Code (SMC)
- Variable Instruction length
- New Instructions and Impact
- Precise exception
- X86 FP – issues and remedy
Self Modify Code (SMC)

• The rule:
  SMC (mov)
  Jmp

• Implications
  – Single pipe architecture – combined (I and D) caches
  – Harvard architecture
  – Speculative (Branch predication) architecture

• What is it good for?
  – Java?
Variable Instruction length

- Advantages
  - Dense instruction footprint
  - ?
- Disadvantages
  - Superscalar
- Solutions
  - Bit per byte (end of instruction marker)
  - Length decoding
"Brute force" Instruction Decode

Superscalar RISC Decoder

Superscalar CISC Decoder (Wide decoder)

Unknown starting locations: Need to decode all possibilities!
Instruction Decode

Superscalar CISC Decoder (Annotated Cache)
New Instructions and Impact

• The power bag
  – Many unused instructions
  – Segmentation

• Examples
  – MMX
  – SSE1
  – SSE2

• What’s next?
Precise exception

- Advantages
- Commit stage
- Check point
X86 FP – issues and remedy

- FP Stack architecture
- SSEs additions
- X86-64bit architecture
INSTRUCTION SUPPLY
Branch Prediction

– Branch Prediction
– One Level (Bimodal) Branch Predictor
– Two level (Local) Branch Predictor
– Global Branch Predictor
– Combining Branch Predictors
– Advanced Predictors
– Predictors in Commercial Processors

Contributors:
Lihu Rappoport
Jared Stark
Ronny Ronen
References

- “Computer Architecture A Quantitive Approach” Hennessy, Patterson
- “A Study of Branch Prediction Strategies”, J.E. Smith, ISCA81
- “Two Level Adaptive Training Branch Prediction”, Tse-Yu Yeh, Yale Patt, January 1992
- “Trading Conflict and Capacity Aliasing in Conditional Branch Predictors”, Michaud, Seznec, Uhlig. ISCA ‘97
- “The YAGS Branch Prediction Scheme”, Eden, Mudge – 1998
- “Dynamic Branch Prediction with Perceptrons” Jimenez, Lin, Jan 2000 Piecewise Linear Branch Prediction, Jimenez, ISCA 32, June 2005
- “TaGe - A case for (partially) TAgged GEometric history length branch prediction”, André Seznec and Pierre Michaud, JILP Vol 8, February, 2006
Introduction

• Microprocessors require accurate branch prediction
  – Improved ILP ⇒ Out-Of-Order Execution ⇒ large ROB
    » Branch every ~5 instructions ⇒ Speculative Execution
    » Deep/wide pipe ⇒ branch misprediction penalty

• **Target:** Maximize correct branch prediction rate for a given predictor size

• Need to predict:
  – Conditional branch *direction*.
  – All branches *targets*. 
The Generic Processor

- Instruction Cache
- BTB
- Instruction fetch/decode
- Rename
- Scheduler
- Trace/decoded cache
- Data supply
- Execution engine
Branch Types

• **Conditional / Unconditional**
  - **Conditional:**
    - Need to predict both: **direction and target address**
    - Actual direction is known only after execution
    - Wrong direction prediction causes a **full flush of pipelines**
  - **Unconditional:**
    - Need to predict **only target address**

• **Direct / Indirect**
  - **Direct:**
    - Target address specified within instruction (as an immediate value)
    - Actual target address is known after decode
    - Wrong target prediction causes a **short flush**
  - **Indirect:**
    - Target address needs to be computed
    - Actual target address is known after execution
    - Wrong target prediction causes a **full flush**

---

Typical Distribution in X86

<table>
<thead>
<tr>
<th></th>
<th>Cond</th>
<th>Un cond</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>70%</td>
<td>25%</td>
</tr>
<tr>
<td>indir</td>
<td>0%</td>
<td>5%</td>
</tr>
</tbody>
</table>
Branches and Performance

- Misprediction rate – miss-per-branch
- MPI : Miss-Per-Instruction:

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

- MPI correlates well with performance.
  For example:
  - MPI = 1% (1 out of 100 instructions \(\cong\) 1 out of 20 branches)
  - IPC=2 (Instructions Per Cycle),
  - flush penalty of 10 cycles.
- We get:
  \(\Rightarrow\) MPI = 1% \(\Rightarrow\) flush in every 100 instructions
  \(\Rightarrow\) flush in every 50 cycles (since IPC=2),
  \(\Rightarrow\) 10 cycles flush penalty every 50 cycles
  \(\Rightarrow\) 20% in performance.
Performance impact of Branch Predictor in Out of Order machine
**Target Array (TA) Prediction**

- TA is accessed using the Instruction Pointer address (hit → branch)
- Implemented as an $n$-way set associative cache
- Tags usually partial
  - Save space
  - Few branches aliased to the same entry
  - BP content does not have to be correct
    
    **Target is performance improvement**

- TA predicts the following
  - Indication that instruction is a branch
  - Predicted target
  - Branch type
    » Unconditional: take target
    » Conditional: predict direction

- TA allocated / updated at execution

![Diagram of Target Array (TA) Prediction](attachment:diagram.png)
Target Array + Direction Prediction

- Target address and direction are predicted separately

![Diagram showing target array and direction predictor](chart)

- Branch IP
- Target address and direction prediction
- Direction predictor
- Hit/miss (indicates a branch)
- Predicted target
- Predicted type
- Predicted direction (taken/not-taken)
Conditional Branch Direction Predictor
One-Bit Predictor

Problem: 1-bit predictor has a double mistake in loops

Branch Outcome: 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1
Prediction: ? 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0
Bimodal (2-bit) Predictor

- A 2-bit counter avoids the double mistake in glitches
  - Need “stronger evidence” to change prediction
- 2 bits encode one of 4 states
  - 00 – Strong NT, 01 – Weakly NT, 10 – Weakly taken, 11 – Strong taken
  - Initial state makes impact: example; Weakly-Taken (most branches are Taken)
- Update
  - Branch was actually taken: increment counter (saturate at 11)
  - Branch was actually not-taken: decrement counter (saturate at 00)
- Predict according to m.s.bit of counter (0 – NT, 1 – taken)

- Predicts well monotonic branches: one mistake per loop invocation
- Does not predict well branches with patterns like 0101…01
BTB (cont’') One Level Prediction

- History/Prediction algorithm

**One bit**

**Two bits**

“A study of Branch Prediction Strategies” J. E. Smith, ISCA81
Bimodal Predictor (cont.)

2-bit-saturate counter array

branch IP  \[\rightarrow\] Prediction* = msb of counter

Update counter with branch outcome

* Exception in A1 automaton
Bimodal Predictor - example

- Example: Br1 prediction
  - Pattern: 1 0 1 0 1 0
  - counter: 2 3 2 3 2 3
  - Prediction: T T T T T T

- Example: Br2 prediction
  - Pattern: 0 1 0 1 0 1
  - counter: 2 1 2 1 2 1
  - Prediction: T nT T nT T nT

- Example: Br3 prediction
  - Pattern: 1 1 1 1 1 0
  - counter: 2 3 3 3 3 3
  - Prediction: T T T T T T

Code:

→ Loop: ....
→ br1: if (n/2) {
→ ......... }
→ br2: if ((n+1)/2) {
→ ......... }
→ n--
→ br3: JNZ n, Loop
Local Predictor

• Local Predictor: prediction made based on a specific Branch history

• The history (sequence) of each branch is saved in a Branch History Register (BHR):

  • BHR: a shift-register updated by branch outcome
  • The BHR points to an array of 2-bit saturated counters
  • Each counter predicts the branch for a given history sequence
    ⇒ may predict well also complicated patterns
    ⇒ Very good for repeated patterns like loops

• The saturated counters may be either
  – private: per Branch
  – per-set: shared by all Branches in the same set
  – global: shared by all Branches

• Too long BHRs are not good:
  – Past history may be no longer relevant
  – Warm-Up is longer
  – Large structure
Local Predictor (cont.)

- In one bit “counter” there could be glitches from the pattern →
  Use 2-bit saturating counters to record outcome:

  - Update counter with branch outcome
  - History with branch outcome
  - Prediction = msb of counter

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

Each BHR bit pattern points to a counter

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

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

\[ \text{size} = 1024 \times (8 + 6 + 2 \times 2^6) = 142\text{Kbit} \]

Remark: \# of Branch IPs = \# of entries in BTB
Local Predictor: example

- The branch of the inner loop in:

```c
for (j = 0; j < 1000; j++)
    for (i = 0; i < 4; i++)
```

may generate the sequence

```
000100010001 . . .
```

- Assuming a history of length 6, at the steady-state, the following patterns will be repeated:

```
000100010001 . . .
```

- The 2 bit counter pointed by 000100, 010001, 100010 should predict next branch to be 0
- The 2 bit counter pointed by 001000 should predict 1
Local Predictor: shared counter arrays

- Local predictor issue: Predictor size too large
- Remedy: use a single counter array shared by all BHR’s
  - All BHR’s index the same array (same history pattern points to the same 2 bit counter)
  - Problem: Branches with similar patterns interfere with each other

Example: #BHRs = 1024; tag_size=8; history size=6

\[
\text{size} = 1024 \times (8 + 6) + 2 \times 2^6 = 14.1 \text{Kbit}
\]
Local Predictor: *lselect*

- **Shared counter array issue**: Branches with similar patterns interfere with each other.
- **Remedy**: *lselect* - reduces inter-branch-interference in the counter array.
- **Inefficient usage of storage**: Table may be sparse because not all histories interfere.

### Diagram

- **Branch IP**
- **2-bit-sat counter array**
- **Prediction** = msb of counter

For $m$ size = 0  ➞ shared counter array predictor
For $m$ size = $\log_2$ (Number of Branch IPs)  ➞ Private counter array

**Predictor size**: $\#BHRs \times [\text{tag}_\text{size} + h] + 2 \times 2^{h + m}$
Local Predictor: *Ishare*

- **Iselect issues**: Inefficient usage of storage
- **Remedy**: *Ishare* reduces inter-branch-interference in the counter array by mapping common patterns in different branches to different counters

```
prediction = msb of counter
```
Global Predictor

Global Predictor:
History cache is a collection of ALL branch results without reference to their IP

- Occasionally, the behavior of some branches is highly correlated with the behavior of other branches:
  
  if (x < 1) . . .
  if (x > 1) . . .

- Using a Global History Register (GHR), the prediction of the second if may be based on the direction of the first if

- For other branches the history interference might be destructive

- History interference may be reduced by using gshare or gselect
Global Predictor (cont.)

All Branch IPs points at same GHR

The predictor size: \[ \text{history}_\text{size} + 2 \times 2^{\text{history}_\text{size}} \]

Example: \[ \text{history}_\text{size} = 12 \Rightarrow \text{size} = 8 \text{ K Bits} \]
Global Predictor: Gshare

_gshare_ combines the global history information with the branch IP

2-bit-sat counter array

GHR

Branch IP

Update History with branch outcome

prediction = msb of counter

Update counter with branch outcome

_gselect_ concatenates global history with the branch IP (not shown): each branch IP X GHR value has a counter (state)
A chooser selects between 2 basic predictors that predict the same branch:
- Use the predictor that was more correct in the past.
Chooser (cont’)

- A chooser enables to decide by which of two predictors each branch is predicted best
- Each branch is predicted by both predictors: \( P1 \) and \( P2 \)
- The branch IP also indexes a 2-bit-sat counter in the chooser array:
  - If \( P1 \) is correct and \( P2 \) is wrong the counter is incremented
  - If \( P2 \) is correct and \( P1 \) is wrong the counter is decremented
  - Otherwise, the counter is not changed
- Next time the branch encountered, it is predicted according to the predictors selected by the chooser
A **chooser** selects between 2 predictor that predict the same branch

- Use the predictor that was more correct in the past

**The chooser may also be indexed by the GHR**

**Many combinations of predictors and choosers are possible**

**Chooser array (an array of 2-bit sat. counters)**

+1 if Bimodal / Local correct and Global wrong
-1 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 specific branch may see stale history
    - Global: future occurrences of all branches may see stale history
  - If History is speculatively updated according to the prediction
    $\Rightarrow$ History must be corrected if the branch is mispredicted
    $\Rightarrow$ 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
  - Counter array is not updated speculatively: prediction can change (state change from 01 to 10 or 10 to 01) only on a misprediction

- **On branch resolution**
  - Update the real history and reset speculative histories if mispredicted
Return Stack Buffer

- A **return** instruction is a special case of an indirect branch:
  - Each times it jumps to a different target
  - The target is determined by the location of the corresponding call instruction
- The idea:
  - Hold a small stack of targets
  - On each call instruction push the address of the instruction which follows the call into the stack
  - On each return pop a target from the stack and load the PC with that target.
  - Both done on fetch!
- A **real indirect branch**:
  - Use local or global history to predict target.
  - Verify at execution
  - See Pentium-M later
Confidence Level Prediction
Confidence Level Prediction

• A confidence level predictor (CLP)
  – Provides a *confidence value* for predictions made by a predictor
  – *Confidence* value estimates the probability for correct prediction
• Confidence level is attached to conditional branches
• We are looking at *binary* CLP
  – Conditional branches are classified into two sub-groups:
    » The group of *high confidence* branches
    » The group of *low confidence* branches
Why Confidence Level

- **Speculation**
  - In a resource limited processor, execute only speculative code which has a high confidence to commit

- **Multi-threading**
  - Choose which thread to execute in a multithreading processor

- **Eager execution**
  - Trigger eager execution for low confidence branches only

- **Power saving**
  - In a power limited processor, execute only speculative code which has a high confidence to commit

- **Insertion into fast misprediction recovery mechanism**

- **Pollution avoidance**
  - Deallocate low confidence static branch from BTB

- **Prediction reversal**
Confidence Level Information

- Static information
  - e.g.,: backward-forward, branch type, target distance
  - Available from the branch instruction itself
    » Unknown at prediction time
  - Costs nothing, predictor-independent
- Predictor information
  - e.g.: the state used for making the prediction (weak/strong)
  - Available from the predictor, Predictor-dependent
  - Gathered and saved using extra hardware, Costs space (and power)
  - May be either predictor-dependent or predictor-independent
Static Information

• The forward / backward direction of the branch
  – Known from the branch displacement
    » Since conditional branches are IP relative, no calculations required
  – Most forward branches are not taken
    » If a forward branch is predicted taken, confidence is lower
  – Most backward branches are taken
    » If a backward branch is predicted not taken, confidence is lower
Predictor information

- Prediction provided by a state machine
  - The state of the state machine used for the prediction (weak/strong)

- In a multi-stage predictor (like chooser)
  - How many of the predictors (of all the predictors which predict) agree with the prediction.

- For a history based predictor (either local or global)
  - The $n$ most recent bits of the prediction history