Review: Branch Target Buffer (BTB)

- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only predicted taken branches and jumps held in BTB
- Next PC determined before branch fetched and decoded

Two-Level Branch Predictor (e.g. GAs)

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)
Accuracy of Different Schemes

- 4096 Entries 2-bit BHT
- Unlimited Entries 2-bit BHT
- 1024 Entries (2,2) BHT

BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table

- 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
  - For SPEC92, 4096 about as good as infinite table

- How could HW predict “this loop will execute 3 times” using a simple mechanism?
  - Need to track history of just that branch
  - For given pattern, track most likely following branch direction

- Leads to two separate types of recent history tracking:
  - GBHR (Global Branch History Register)
  - PABHR (Per Address Branch History Table)

- Two separate types of Pattern tracking
  - GPHT (Global Pattern History Table)
  - PAPHT (Per Address Pattern History Table)

Yeh and Patt classification

- GAg: Global History Register, Global History Table
- PAg: Per-Address History Register, Global History Table
- PAp: Per-Address History Register, Per-Address History Table

Two-Level Adaptive Schemes:

- History Registers of Same Length (6 bits)

- PAp best: But uses a lot more state!
- GAg not effective with 6-bit history registers
  - Every branch updates the same history register
- PAg performs better because it has a branch history table
Versions with Roughly same accuracy (97%)

- Cost:
  - GAg requires 18-bit history register
  - PAg requires 12-bit history register
  - PAp requires 6-bit history register
- PAg is the cheapest among these

Why doesn’t GAg do better?

- Difference between GAg and both PA variants:
  - GAg tracks correlations between different branches
  - PAg/PAp track correlations between different instances of the same branch
- These are two different types of pattern tracking
  - Among other things, GAg good for branches in straight-line code, while PA variants good for loops
- Problem with GAg? It aliases results from different branches into same table
  - Issue is that different branches may take same global pattern and resolve it differently
  - GAg doesn’t leave flexibility to do this

Other Global Variants: Try to Avoid Aliasing

- GAs: Global History Register, Per-Address (Set Associative) History Table
- GShare: Global History Register, Global History Table with Simple attempt at anti-aliasing

Branches are Highly Biased

- From: “A Comparative Analysis of Schemes for Correlated Branch Prediction,” by Cliff Young, Nicolas Gloy, and Michael D. Smith
- Many branches are highly biased to be taken or not taken
  - Use of path history can be used to further bias branch behavior
- Can we exploit bias to better predict the unbiased branches?
  - Yes: filter out biased branches to save prediction resources for the unbiased ones
Exploiting Bias to avoid Aliasing: Bimode and YAGS

Is Global or Local better?

Administrative

- Midterm I: Wednesday 3/16
  Location: 320 Soda Hall
  TIME: 2:30-5:30
  - Can have 1 sheet of 8½x11 handwritten notes – both sides
  - No microfiche of the book!
- This info is on the Lecture page (has been)
- Meet at LaVal’s afterwards for Pizza and Beverages
  - Great way for me to get to know you better
  - I’ll Buy!

Dynamically finding structure in Spaghetti

- Consider complex “spaghetti code”
- Are all branches likely to need the same type of branch prediction?
  - No.
- What to do about it?
  - How about predicting which predictor will be best?
  - Called a “Tournament predictor”

\[ \text{Neuner: some branches local, some global} \]
  - Difference in predictability quite significant for some branches!
Tournament Predictors

- Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved.
- Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector.
- Use the predictor that tends to guess correctly.

Predictor A

Predictor B

addr

history

Tournament Predictor in Alpha 21264

1. 4K 2-bit counters to choose from among a global predictor and a local predictor.
2. Global predictor (GAg):
   - 4K entries, indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor.
   - 12-bit pattern: ith bit 0 => ith prior branch not taken; ith bit 1 => ith prior branch taken.
3. Local predictor consists of a 2-level predictor (PAg):
   - Top level: a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
   - Next level: Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction.

Total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K bits! (~180,000 transistors)

% of predictions from local predictor in Tournament Scheme

Accuracy of Branch Prediction

- Profile: branch profile from last execution (static in that it is encoded in instruction, but profile)

fig 3.40
Review: Memory Disambiguation

- Question: Given a load that follows a store in program order, are the two related?
  - Trying to detect RAW hazards through memory
  - Stores commit in order (ROB), so no WAR/WAW memory hazards.
- Implementation
  - Keep queue of stores, in program order
  - Watch for position of new loads relative to existing stores
  - Typically, this is a different buffer than ROB!
    » Could be ROB (has right properties), but too expensive
- When have address for load, check store queue:
  - If any store prior to load is waiting for its address
    » If load address matches earlier store address (associative lookup),
      then we have a memory-induced RAW hazard:
      » store value available \( \Rightarrow \) return value
      » store value not available \( \Rightarrow \) return ROB number of source
  - Otherwise, send out request to memory
- Will relax exact dependency checking in later lecture

In-Order Memory Queue

- Execute all loads and stores in program order
  
  => Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

  - Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

Conservative O-o-O Load Execution

```
st r1, (r2)
ld r3, (r4)
```

- Split execution of store instruction into two phases: address calculation and data write
- Can execute load before store, if addresses known and \( r4 \neq r2 \)
- Each load address compared with addresses of all previous uncommitted stores (can use partial conservative check i.e., bottom 12 bits of address)
- Don’t execute load if any previous store address not known

(MIPS R10K, 16 entry address queue)
Address Speculation

\[
\text{st } r1, (r2) \\
\text{ld } r3, (r4)
\]

- Guess that \( r4 \neq r2 \)
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find \( r4 = r2 \), squash load and all following instructions

\( \Rightarrow \) Large penalty for inaccurate address speculation

Memory Dependence Prediction (Alpha 21264)

\[
\text{st } r1, (r2) \\
\text{ld } r3, (r4)
\]

- Guess that \( r4 \neq r2 \) and execute load before store
- If later find \( r4 = r2 \), squash load and all following instructions, but mark load instruction as \text{store-wait}
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear \text{store-wait} bits

Speculative Loads / Stores

Just like register updates, stores should not modify the memory until after the instruction is committed
- A speculative store buffer is a structure introduced to hold speculative store data.

Speculative Store Buffer

- On store execute:
  - mark entry valid and speculative, and save data and tag of instruction.
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit
Speculative Store Buffer

- If data in both store buffer and cache, which should we use: Speculative store buffer
- If same address in store buffer twice, which should we use: Youngest store older than load

Memory Dependence Prediction

- Important to speculate?
  - Two Extremes:
    - Naive Speculation: always let load go forward
    - No Speculation: always wait for dependencies to be resolved
- Compare Naive Speculation to No Speculation
  - False Dependency: wait when don’t have to
  - Order Violation: result of speculating incorrectly
- Goal of prediction:
  - Avoid false dependencies and order violations

<table>
<thead>
<tr>
<th>Program</th>
<th>Naive Speculation</th>
<th>No Speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec</td>
<td>Memory</td>
<td>Order</td>
</tr>
<tr>
<td>go</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>mSlics</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>gcc</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>congress</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>alhpc</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>Qspec</td>
<td>23</td>
<td>13</td>
</tr>
<tr>
<td>perl_prm</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>perl_scrab</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>cortex</td>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>tomcat</td>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>swim</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>mgrid</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>upp</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>apsi</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>gppp</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>wave5</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>turb3d</td>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

From “Memory Dependence Prediction using Store Sets”, Chrysofs and Emer.

Premise: Past indicates Future

- Basic Premise is that past dependencies indicate future dependencies
  - Not always true! Hopefully true most of time
- Store Set: Set of store insts that affect given load
  - Example: Addr Inst
    - 0 Store C
    - 4 Store A
    - 8 Store B
    - 12 Store C
    - 28 Load B → Store set { PC 8 }
    - 32 Load D → Store set { (null) }
    - 36 Load C → Store set { PC 0, PC 12 }
    - 40 Load B → Store set { PC 8 }
  - Idea: Store set for load starts empty. If ever load go forward and this causes a violation, add offending store to load's store set
- Approach: For each indeterminate load:
  - If Store from Store set is in pipeline, stall
  Else let go forward
- Does this work?

Said another way: Could we do better?

- Results from same paper: performance improvement with oracle predictor
  - We can get significantly better performance if we find a good predictor
  - Question: How to build a good predictor?
How well does an infinite tracking work?

• “Infinite” here means to place no limits on:
  – Number of store sets
  – Number of stores in given set

• Seems to do pretty well
  – Note: “Not Predicted” means load had empty store set
  – Only Applu and Xlisp seems to have false dependencies

How to track Store Sets in reality?

• SSIT: Assigns Loads and Stores to Store Set ID (SSID)
  – Notice that this requires each store to be in only one store set!
• LFST: Maps SSIDs to most recent fetched store
  – When Load is fetched, allows it to find most recent store in its store set that is executing (if any) ⇒ allows stalling until store finished
    » Pretty much same type of ordering as enforced by ROB anyway
    » Transitivity ⇒ loads end up waiting for all active stores in store set

  – When Store is fetched, allows it to wait for previous store in store set
  – Only Applu and Xlisp seems to have false dependencies

What if store needs to be in two store sets?

• Allow store sets to be merged together deterministically
  – Two loads, multiple stores get same SSID
• Want periodic clearing of SSIT to avoid:
  – problems with aliasing across program
  – Out of control merging

How well does this do?

• Comparison against Store Barrier Cache
  – Marks individual Stores as “tending to cause memory violations”
  – Not specific to particular loads....

• Problem with APPLU?
  – Analyzed in paper: has complex 3-level inner loop in which loads occasionally depend on stores
  – Forces overly conservative stalls (i.e. false dependencies)

Load Value Predictability

• Try to predict the result of a load before going to memory
• Paper: “Value locality and load value prediction”
  – Mikko H. Lipasti, Christopher B. Wilkerson and John Paul Shen

• Notion of value locality
  – Fraction of instances of a given load that match last n different values

• Is there any value locality in typical programs?
  – Yes!
    – With history depth of 1: most integer programs show over 50% repetition
    – With history depth of 16: most integer programs show over 80% repetition
    – Not everything does well: see cjpeg, swm256, and tomcatv

• Locality varies by type:
  – Quite high for inst/data addresses
  – Reasonable for integer values
  – Not as high for FP values
Load Value Prediction Table (LVPT)

- Untagged, Direct Mapped
- Takes instructions \( \Rightarrow \) Predicted Data

- Contains history of last n unique values from given instruction
- Can contain aliases, since untagged

- How to predict?
  - When \( n = 1 \), easy
  - When \( n = 16 \)? Use Oracle

- Is every load predictable?
  - No! Why not?
  - Must identify predictable loads somehow

Load Classification Table (LCT)

- Untagged, Direct Mapped
- Takes instructions \( \Rightarrow \) Single bit of whether or not to predict

- How to implement?
  - Uses saturating counters (2 or 1 bit)
  - When prediction correct, increment
  - When prediction incorrect, decrement

- With 2 bit counter
  - 0,1 \( \Rightarrow \) not predictable
  - 2 \( \Rightarrow \) predictable
  - 3 \( \Rightarrow \) constant (very predictable)

- With 1 bit counter
  - 0 \( \Rightarrow \) not predictable
  - 1 \( \Rightarrow \) constant (very predictable)

Accuracy of LCT

- Question of accuracy is about how well we avoid:
  - Predicting unpredictable load
  - Not predicting predictable loads

- How well does this work?
  - Difference between “Simple” and “Limit”: history depth
    » Simple: depth 1
    » Limit: depth 16
  - Limit tends to classify more thing as predictable (since this works more often)

- Basic Principle:
  - Often works better to have one structure decide on the basic “predictability” of structure
  - Independent of prediction structure

Constant Value Unit

- Idea: Identify a load instruction as “constant”
  - Can ignore cache lookup (no verification)
  - Must enforce by monitoring result of stores to remove “constant” status

- How well does this work?
  - Seems to identify 6-18% of loads as constant
  - Must be unchanging enough to cause LCT to classify as constant
Load Value Architecture

- LCT/LVPT in fetch stage
- CVU in execute stage
  - Used to bypass cache entirely
  - (Know that result is good)
- Results: Some speedups
  - 21264 seems to do better than Power PC
  - Authors think this is because of small first-level cache and in-order execution makes CVU more useful

Review: Memory Disambiguation

- Question: Given a load that follows a store in program order, are the two related?
  - Trying to detect RAW hazards through memory
  - Stores commit in order (ROB), so no WAR/WAW memory hazards.
- Implementation
  - Keep queue of stores, in program order
  - Watch for position of new loads relative to existing stores
  - Typically, this is a different buffer than ROB!
    - Could be ROB (has right properties), but too expensive
- When have address for load, check store queue:
  - If any store prior to load is waiting for its address
    - ???
  - If load address matches earlier store address (associative lookup), then we have a memory-induced RAW hazard:
    - store value available \( \Rightarrow \) return value
    - store value not available \( \Rightarrow \) return ROB number of source
  - Otherwise, send out request to memory
- Will relax exact dependency checking in later lecture

Memory Dependence Prediction

- Important to speculate?
  - Naïve Speculation: always let load go forward
  - No Speculation: always wait for dependencies to be resolved
- Compare Naïve Speculation to No Speculation
  - False Dependency: wait when don’t have to
  - Order Violation: result of speculating incorrectly
- Goal of prediction:
  - Avoid false dependencies and order violations

<table>
<thead>
<tr>
<th>Spec95 Program</th>
<th>Naïve Speculation</th>
<th>No Speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory Viol./Per</td>
<td>Memory Trap</td>
</tr>
<tr>
<td></td>
<td>1K Insts</td>
<td>Penalty (Cycles)</td>
</tr>
<tr>
<td>go</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>m88kem</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>gss</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>compress</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>xlfop</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>fjew</td>
<td>23</td>
<td>15</td>
</tr>
<tr>
<td>perl prim</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>perf scrun</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>vortex</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>tomcatv</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>swim</td>
<td>2</td>
<td>36</td>
</tr>
<tr>
<td>mgrid</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>apgl</td>
<td>18</td>
<td>22</td>
</tr>
<tr>
<td>apsl</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>ftpper</td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>wave5</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>turb2d</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

From “Memory Dependence Prediction using Store Sets”, Chrysos and Emer.

Said another way: Could we do better?

- Results from same paper: performance improvement with oracle predictor
  - We can get significantly better performance if we find a good predictor
  - Question: How to build a good predictor?
Premise: Past indicates Future

- Basic Premise is that past dependencies indicate future dependencies
  - Not always true! Hopefully true most of time
- Store Set: Set of store.insts that affect given load
  - Example:
    | Addr | Inst | Store Set |
    |------|------|-----------|
    | 0    |      | Store C   |
    | 4    |      | Store A   |
    | 8    |      | Store B   |
    | 12   |      | Store C   |
  - Example: 
    | Addr | Inst | Store Set |
    |------|------|-----------|
    | 28   |      | Store set { PC 8 } |
    | 32   |      | Store set { null } |
    | 36   |      | Store set { PC 0, PC 12 } |
    | 40   |      | Store set { PC 8 } |
  - Idea: Store set for load starts empty. If ever load go forward and this causes a violation, add offending store to load’s store set
- Approach: For each indeterminate load:
  - If Store from Store set is in pipeline, stall
  - Else let go forward
- Does this work?

How well does an infinite tracking work?

- “Infinite” here means to place no limits on:
  - Number of store sets
  - Number of stores in given set
- Seems to do pretty well
  - Note: “Not Predicted” means load had empty store set
  - Only Applu and Xlisp seems to have false dependencies

How to track Store Sets in reality?

- SSIT: Assigns Loads and Stores to Store Set ID (SSID)
  - Notice that this requires each store to be in only one store set!
- LFST: Maps SSIDs to most recent fetched store
  - When Load is fetched, allows it to find most recent store in its store set that is executing (if any) ⇒ allows stalling until store finished
  - When Store is fetched, allows it to wait for previous store in store set
    » Pretty much same type of ordering as enforced by ROB anyway
    » Transitivity ⇒ loads end up waiting for all active stores in store set
- What if store needs to be in two store sets?
  - Allow store sets to be merged together deterministically
    » Two loads, multiple stores get same SSID
- Want periodic clearing of SSIT to avoid:
  - problems with aliasing across program
  - Out of control merging

How well does this do?

- Comparison against Store Barrier Cache
  - Marks individual Stores as “tending to cause memory violations”
  - Not specific to particular loads…..
- Problem with APPLU?
  - Analyzed in paper: has complex 3-level inner loop in which loads occasionally depend on stores
  - Forces overly conservative stalls (i.e. false dependencies)
Conclusion

• Two-Level Branch Prediction
  – Uses complex history (either global or local) to predict next branch
  – Two tables: a history table and a pattern table
  – Global Predictors: GAg, GAs, GShare, Bimode, YAGS
  – Local Predictors: PAg, PAp, PAs

• Dependence Prediction: Try to predict whether load
  depends on stores before addresses are known
  – Store set: Set of stores that have had dependencies with load in past

• Last Value Prediction
  – Predict that value of load will be similar (same?) as previous value
  – Works better than one might expect

• Dependence Prediction: Try to predict whether load
  depends on stores before addresses are known
  – Store set: Set of stores that have had dependencies with load in past