Branch Prediction and Trace Cache

PSU ECE587
Reducing branch costs with dynamic hardware prediction

- Branch prediction basics:
  - We need to predict conditional branch outcome to select the address for next instruction fetch
    - PC + 4
    - Or branch target address
  - Also we need to quickly determine the branch target address
    - Direct branches
    - Register indirect branches
    - Returns
Predicting conditional branch outcomes

- Simplest dynamic branch prediction scheme is a *branch-prediction buffer* or *branch history table*.
- It is a small memory indexed by the lower portion of the branch address.
- Stores previous branch outcomes to predict next outcome.
- Memory is not *tagged*.
- Prediction may have been put in the entry by a different branch.
Predicting conditional branch outcomes

- 1-bit prediction buffer stores the last executed branch outcome, and uses it to predict the next outcome
  - If bit = 1, branch is predicted taken
  - If bit = 0, branch is predicted not-taken

- A simple 1-bit scheme has a performance shortcoming
  - A series of branch outcomes and corresponding predictions:
    - outcomes: 1111011110111101
    - predictions: 1111011110111100
    - mispredictions: 111011110111110
Predicting conditional branch outcomes

- 2-bit saturating counter often used
  - Branch taken $\rightarrow$ increment state
    - Max state “11” stays at “11” when incremented
  - Branch not-taken $\rightarrow$ decrement state
    - Min state “00” stays at “00” when decremented
  - “11” and “10” are predict taken states
  - “00” and “01” are predict not-taken states
2-bit saturating counter state-machine

- Predict taken “11”
  - T
  - NT

- Predict taken “10”
  - T
  - NT

- Predict not taken “00”
  - T
  - NT

- Predict not taken “01”
  - T
  - NT
Assuming initial state to be “11”, branch outcomes and corresponding predictions now look as follows:

<table>
<thead>
<tr>
<th>outcomes</th>
<th>1111011110111101</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>333323333233332</td>
</tr>
<tr>
<td>predictions</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>mispredictions</td>
<td>1111111111111111</td>
</tr>
</tbody>
</table>
Correlating branch predictors

- 2-bit prediction schemes use the recent behavior of a single branch to predict the future behavior of that branch.
- Behavior of longer sequence of branch execution history often provides more accurate prediction outcome.
- Behavior of other branches rather than just the branch we are trying to predict is sometimes important:
  - Because outcomes of different branches often correlate.
  - Global branch history.
- For some branches, prior history execution of the branch is important:
  - Because of loops.
  - Local branch history.
Correlating branch predictors: code example

```c
if (aa == 2) {
    aa = 0;
}
if (bb == 2) {
    bb = 0;
}
if (aa != bb) {
    L1: DSUBUI R3,R3,R1,#2
    BNEZ R3,L1
    DADD R1,R0,R0
    L2: DSUBU R3,R1,R2
    BEQZ R3,L3
```
Correlating branch predictor with 2-bit global history register
Adaptively combining branch predictors

- Some branches are predicted more accurately with *global* predictors
- Other branches are predicted better with *local* predictors
- It is possible to combine both types of predictors, and dynamically select the right predict for the right branch
- The selector is yet another predictor with 2-bit state machine per entry
State machine transitions for a selector
Branch target buffer (BTB)

- It is a cache that stores branch targets.
- It is accessed by the address of the instruction currently fetched.
- Allows branch target to be read in the IF stage.
  - When a branch is predicted taken, the fetch of the instruction at the branch target address can proceed immediately in the next cycle.
  - Stall cycles that would have been needed to wait for the decoding of the branch and the computation of the target are saved.
Branch target buffer

PC of instruction to fetch

Look up

Predicted PC

Number of entries in branch-target buffer

Yes: then instruction is branch and predicted PC should be used as the next PC

No: instruction is not predicted to be branch; proceed normally

Branch predicted taken or untaken
Predicting return address using Return Address Stack (RAS)

- Indirect branches have multiple potential targets, since address comes from a register, which can have many possible values
- Branch target buffers could be used for indirect branch target prediction
  - However, many mispredictions can happen because the BTB can store only one target per branch
- Most indirect branches come from return instructions
Return address stack

- A small address buffer organized as a stack
- When a Call is encountered the Return address (which is Call address + 4) is pushed onto the RAS
- When a Return instruction is encountered, the address from the top of the RAS is popped and used as the target
Trace Cache

- Future high performance requires increased instruction fetch bandwidth, with minimum latency.
- Paper proposes supplementing a conventional instruction cache with a trace cache.
- Non-contiguous instructions appear contiguous in the trace cache.
Trace Cache

- Performance issues emerging as processor issue rates increase:
  - Branch throughput
    - Predicting multiple branches, including taken every cycle.
  - Non-contiguous instructions and alignment.
  - Fetch unit latency.
One possible solution: allow fetch from multiple non-contiguous blocks:

- Multiple addresses have to be generated before fetch begin:
  - Implies a level of indirection.
  - Additional fetch stage latency.
- Multiported or interleaved ICache is required.
- Instruction merging and alignment increase fetch latency.
Trace Cache

Basic idea:

- Conventional instruction cache holds instructions in static program order.
- Trace cache holds instructions in dynamic program order.
- See Figure 2 in the paper.
Trace Cache

- The core fetch unit
  - Interleaved sequential: 2 consecutive blocks can be accessed in the same cycle.
  - Fetch up to the limit or to the next taken branch.
  - Limit of 16 instructions and 3 branches.
  - 16-way interleaved BTB.
  - Multiple branch GAg predictor:
    - Global history register.
    - Rearranged PHT organization: 8 state machines per pattern.
Trace Cache

- Figure 4 shows the core fetch unit and the trace cache.
- Trace cache organization:
  - Up to 16 instructions wide and 3 branches.
  - Contains a fill buffer, instruction traces and control information.
Trace Cache

- Control state:
  - Valid bit
  - Tag to identify trace starting address
  - Branch flags to indicate taken/not-taken direction
  - Branch mask defines number of branches in a trace
  - Trace fall through address
  - Trace target address
  - End of trace marker

- Fill buffer has to stop the trace at indirect branches
Trace Cache

- Trace cache design space
  - Address associativity and path associativity
  - Partial trace matching
  - Indexing method
  - Fill issues, e.g. number of fill buffers
  - Trace selection
    - Some committed traces are never reused
  - Victim trace cache
  - Redundancy
Trace Cache

- Simulation processor model
  - Very large instruction window (2048)
  - Unlimited renaming
  - Unlimited functional units
  - Perfect data cache
  - Perfect memory disambiguation
- Results shown for trace cache and multiple fetch blocks cache organizations of 1, 2 and 3 cycles latency
Reading Assignment