Reducing Design Complexity of the Load/Store Queue

I. Park, C.L. Ooi, T.N. Vijaykumar
What is a Load/Store Queue?

• Two separate queues
  – Load Queue
  – Store Queue

• Functions
  – Buffer and maintain all in-flight memory instructions in program order
  – Support associative searches to honor memory dependence
  – Support associative searches to enforce memory consistency
Scaling techniques

• Reduce search bandwidth demand
  – Store queue
    • Store-set predictor
  – Load queue
    • Load buffer holds all out-of-order-issued loads

• Increase queue capacity
  – Segment Load/Store queue into multiple smaller queues and chain them together
Reducing Store Queue Search

- **Motivation**
  - If we can predict that a load will not find any matching store, we can avoid performing useless store queue search

- **Solution - Store-set predictor**
  - Have a counter keep track of all non-committed stores
  - If counter
    - larger than zero, load must search the store queue
    - zero, load obtains value from cache hierarchy

<table>
<thead>
<tr>
<th>store-set only</th>
<th>store</th>
<th>Fetch</th>
<th>Issue</th>
<th>Commit</th>
<th>Commit stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>store</td>
<td></td>
<td>Valid = true;</td>
<td>Valid = false;</td>
<td>update SSID</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td></td>
<td>update LFST</td>
<td>read Valid</td>
<td>update SSID</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>store-set + store-load pair</th>
<th>store</th>
<th>Fetch</th>
<th>Issue</th>
<th>Commit</th>
<th>Commit stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid = true; Counter++; update LFST</td>
<td>Valid = false;</td>
<td>update SSID</td>
<td>Counter--;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| load | access SSID&LFST | read Valid | read Counter | update SSID |
Reducing Load Queue Search

• Motivation
  – Average number of out-of-order loads is small, so why search Load Queue every time

• Solution – Load Buffer
  – Keep out-of-order-issued loads separate from Load Queue
  – When load issues
    • Oldest non-issued load
      – Search Load Buffer
    • Else
      – Copy load address into Load Buffer
Increasing Queue Capacity

• Motivation
  – If you *have to* search, search less if possible *(not really sure about this)*

• Solution - Segmentation
  – Instruction performs dependence search on its segment first
  – If no match, continue to next segment

• Allocation strategies
  – *no-self-circular* - many segs
  – *self-circular* - compact
Results

• Set-predictor
  – Reduces search bandwidth demand 67%(int), 76%(float)

• Load buffer
  – Reduces search bandwidth demand 74%(int), 77%(float)

• Set-predictor + Load buffer
  – Average speedup 2%(int), 7%(float)

• Segmenting the load/store queue
  – Self-circular outperforms no-self-circular

• Combining it all
  – Average speedup 6%(int), 23%(float)
Questions

• The improvement provided by segmentation is not that impressive. Can it be further enhanced?
• How many ports do today’s load/store queues have?
• Is it still expensive to build multi-port load/store queue?
• Could we adaptively increase or decrease the aggressiveness of our predictors according to their success?
• What would be the effect of fewer hardware guarantees that put more pressure on the software/compiler writers to guarantee that mis-ordering errors will never happen? Would “delay” slots need to be inserted or could they do better?
• How do you decide the benchmark to evaluate the architecture?
• Only two benchmarks are used in the simulation. Do we still have improvements if we use other types of benchmarks?
• If it’s possible that a new approach works well in some benchmarks in experiment, but not well for some others? How to deal with this case?