Analyzing the Benefits of More Complex Cache Replacement Policies in Moderns GPU LLCs

Jarvis (Yuxiao) Jia and Matthew D. Sinclair
University of Wisconsin-Madison
jia44@wisc.edu    sinclair@cs.wisc.edu
Background: m5 + GEMS = gem5

Ruby: more sophisticated & adaptable
• More in-depth coherence support

Classic: quick, simpler option
• Often easier to configure
• Only basic MOESI coherence protocol

<table>
<thead>
<tr>
<th></th>
<th>Ruby</th>
<th>Classic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacement policies</td>
<td>LRU, PseudoLRU</td>
<td>Random, LRU, TreePLRU, BIP, LIP, MRU, LFU, FIFO, Second-Chance, NRU, RRIP, BRRIP</td>
</tr>
<tr>
<td>Coherence protocols</td>
<td>M1_example, MESI_Two_Level, MOESI_CMP_directory, MOESI_CMP_token, MOESI_hammer, MESI Three Level, CHI, …</td>
<td>MOESI (snooping protocol)</td>
</tr>
</tbody>
</table>

Problem: Ruby cannot use state-of-the-art replacement policies in Classic
Merging Replacement Policy Support

- Merged the cache replacement policies from Classic to Ruby
- Users can use any of the replacement policies in either model

How to validate correctness of replacement policies?
Edge case example for SecondChance RP

```python
from m5.objects.ReplacementPolicies import SecondChanceRP as rp

def generator(generator):
    yield generator.create_linear(0, 0, 8, 63, 64, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 128, 191, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 256, 314, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 384, 447, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 6, 63, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 128, 191, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 512, 575, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 640, 782, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 9, 65, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 128, 191, 64, 30000, 30000, 100, 0)
    yield generator.create_linear(0, 0, 10, 8, 8, 30000, 30000, 100, 0)
```


<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way1 | Way2 | Way3 | Way4
-----|------|------|------
A    | 0    |      |      
Miss

Way1  Way2  Way3  Way4

A  0  C  0

Miss

<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Miss

A

C

E

<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>G</td>
</tr>
</tbody>
</table>

Miss

Way1       Way2       Way3       Way4

1          0          0          0

A          C          E          G

Hit

<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>G</td>
</tr>
</tbody>
</table>

Hit

<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>I</td>
<td>G</td>
</tr>
</tbody>
</table>

Miss

<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>I</td>
<td>K</td>
</tr>
</tbody>
</table>

Miss

<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>I</td>
<td>K</td>
</tr>
</tbody>
</table>

Hit

<table>
<thead>
<tr>
<th>Way1</th>
<th>Way2</th>
<th>Way3</th>
<th>Way4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>I</td>
<td>K</td>
</tr>
</tbody>
</table>

Turns Out The Replacement Policies Had Bugs!

- Replacement Policy-specific Bugs (i.e., both Classic and Ruby)
  - 20881: MRU initialized replacement incorrectly
  - 20882: SecondChance initialized new entries incorrectly
  - 65952: FIFO incorrect if multiple new entries in same cycle

- Integration with Ruby-specific Bugs (i.e., only in Ruby)
  - 21099: Ruby called cacheProbe twice in in_ports, causing RP info to be incorrect
  - 62232, 63191, 64371: Ruby updated RP info twice per miss, causing LFU, RRIP, and others to behave incorrectly (MI_example, MESI_Two_Level)
    - This problem may be in other Ruby protocols too

- Current Status: RPs have edge case tests integrated
  - Correctness testing performed as part of gem5 regression testing
How Can We Use These Modern RPs in Ruby?

• Prior work has not examined more complex RPs in GPUs
• Conventional wisdom: LRU sufficient for GPUs
  − Traditional GPGPU workloads have streaming access patterns
  − GPGPU caches traditionally < 64B of space, on average, per thread
  − Thus, unlikely data will remain in caches long enough for RP to matter

Modern GPUs used for an increasingly wide range of applications

These workloads reuse data more frequently

And modern GPUs have increasingly large LLCs

• Added support to use these RPs in gem5’s GPU LLC
Methodology

• System Setup:
  − Vega 20 GPU (60 Compute Units, 16KB L1 D$ per CU)
  − L1 latency: 143, L2 latency: 260, Scalar cache latency: 167
    − Latencies based on Daniel & Vishnu’s GAP work

• Metrics:
  − Vary L2 (LLC) cache sizes: [256KB, 512MB] (powers of two)
  − L2 Replacement Policies: FIFO, LFU, LIP, LRU, MRU, NRU, SRRIP, SecondChance, TreePLRU
  − Write-back and Write-through L2

• Study of mix of streaming and non-streaming workloads:
  − Pannotia, Rodinia
  − Microbenchmarks to better trace access patterns

Show a subset of these results today for brevity
NW (Needleman–Wunsch) WT LLC Execution Time

LFU, MRU generally worse than others – Hurt temporal locality
Little difference between rest of policies until WS fits in LLC
NW (Needleman–Wunsch) WT LLC Hit Rate

LLC Hit Rates confirm performance trends
In general WB LLC caches outperform WT LLC caches – reuse opportunities

Average around 6% less execution time than WT
NW (Needleman–Wunsch) WB LLC Hit Rate

Same RP trends for WT caches (just hit rates vary)
Higher average hit rate than WT
MRU and LFU generally perform worse than other RPs

WB/WT choice seems much more important than RP choice (besides not using MRU/LFU)

Performance affected less by RPs as cache size grows, fixed once WS fits in LLC

Surprising how little RP seems to impact performance
  - Hypothesis: GPU Ruby protocols have similar RP update problems as Ruby CPU protocols
  - Next Step: targeted microbenchmarks with known access patterns
Conclusion

• Classic model has more complex RP support in gem5
  - However, Ruby only supported LRU variants
• We improved gem5’s publicly available RP support
  - Merged RPs – Ruby can now use Classic’s advanced RPs
  - Integrated RP edge case testing into gem5’s regression testing
  - Added support to use these RPs into GPU
• Current Results:
  - MRU and LFU fail to exploit temporal locality (bad choices for GPU)
  - Other RPs provide similar performance to one another
  - WB vs. WT LLC seems to matter a lot more than RP choice
• Next Steps:
  - Use targeted microbenchmarks to debug GPU LLC RP behavior
  - Integrate RP into known good GPU models