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

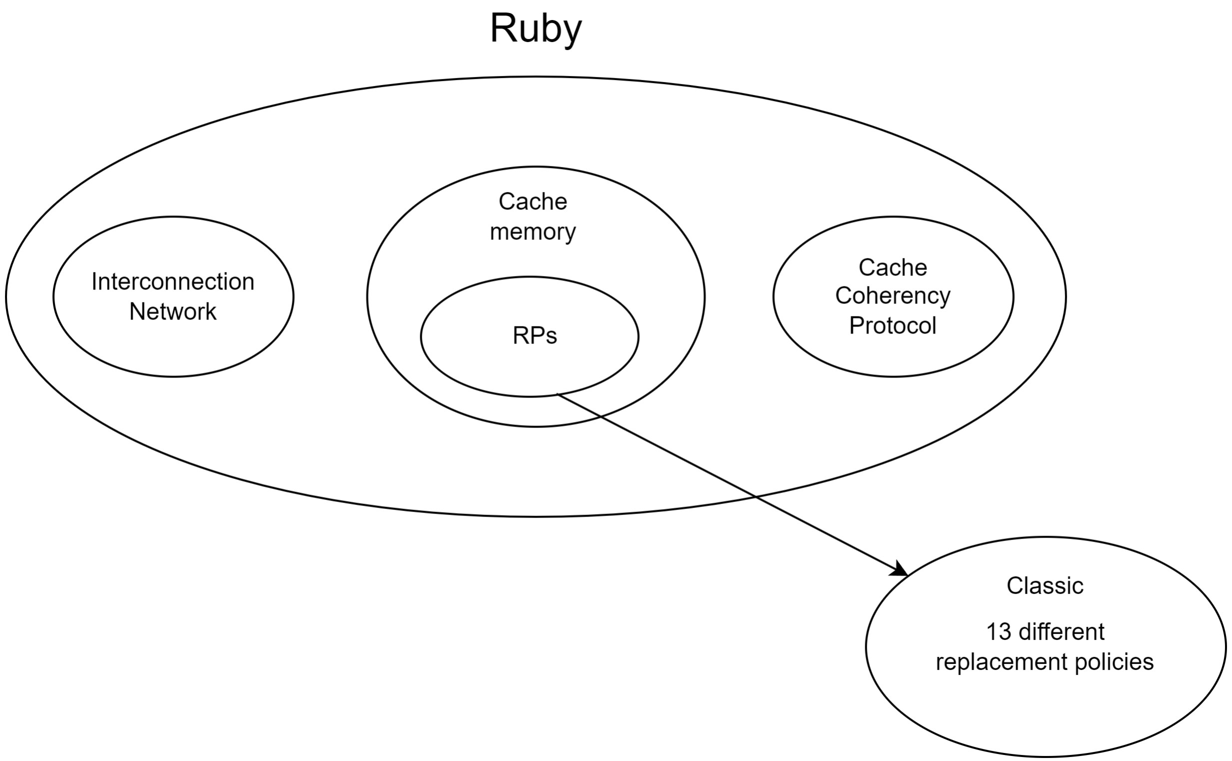
Classic: quick, simpler option

- Often easier to configure
- Only basic MOESI coherence protocol

	Ruby	Classic
Replacement policies	LRU, PseudoLRU	Random, LRU, TreePLRU, BIP, LIP, MRU, LFU, FIFO, Second-Chance, NRU, RRIP, BRRIP
Coherence protocols	MI_example, MESI_Two_Level, MOESI_CMP_directory, MOESI_CMP_token, MOESI_hammer, MESI Three Level, CHI, …	MOESI (snooping protocol)

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

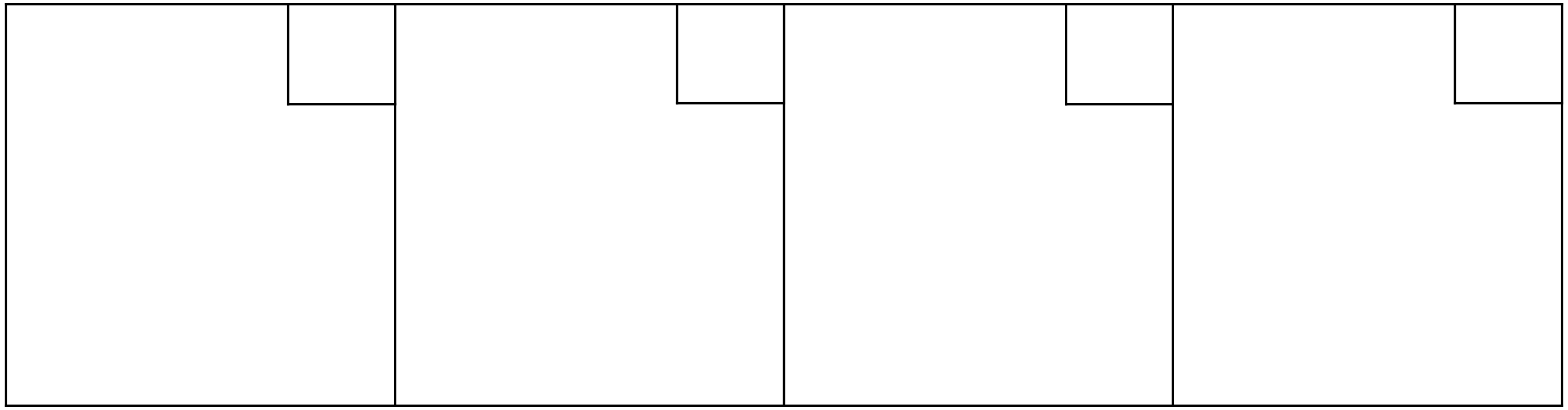
How to validate correctness of replacement policies?

Edge case example for SecondChance RP

```python
# This test is targeting loads.
# Access pattern: A, C, E, G, A, C, I, K, A, C
# Each letter represents a 64-byte address range.

# The [] indicate two different sets, and each set has four ways.
# [set0way0, set0way1, set0way2, set0way3],
# [set1way0, set1way1, set1way2, set1way3],
# If you have a 512B cache with 4-way associativity,
# and each cache line is 64B. This test can be used to test the correctness of Second Chance
# replacement policy. The Second Chance replacement policy will keep the block
# 'A' and 'C' in the cache because of the second chance bit. More specifically,
# with Second Chance replacement policy, you will observe:
# m, m, m, m, h, h, m, m, h, h, where 'm' means miss, and 'h' means hit.

# Explanation of the result:
# The number after each letter is the second chance bit, which would be set after a re-reference.
# A, C, E, G are misses. The cache stores ([A0, C0, E0, G0],[ , , ,]).
# A, C are hit. Now the cache stores ([A1, C1, E0, G0],[ , , ,]).
# I searches a victim and selects E. Now the cache stores ([A0, C0, I0, G0],[ , , ,]).
# K searches a victim and selects G. Now the cache stores ([A0, C0, I0, K0],[ , , ,]).
# A, C are hits.

from m5.objects.ReplacementPolicies import SecondChanceRP as rp


def python_generator(generator):
    yield generator.createLinear(60000, 0, 63, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 128, 191, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 256, 319, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 384, 447, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 0, 63, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 128, 191, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 512, 575, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 640, 703, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 0, 63, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(60000, 128, 191, 64, 30000, 30000, 100, 0)
    yield generator.createLinear(30000, 0, 0, 0, 30000, 30000, 100, 0)

    yield generator.createExit(0)
```

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1 **Way2** **Way3** **Way4**



Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
0			
A			

Miss

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
0	0		
A	C		

Miss

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
0	0	0	
A	C	E	

Miss

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
0	0	0	0
A	C	E	G

Miss

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
1	0	0	0
A	C	E	G

Hit

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
1	1	0	0
A	C	E	G

Hit

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
0	0	0	0
A	C	I	G

Miss

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
0	0	0	0
A	C	I	K

Miss

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
1	0	0	0
A	C	I	K

Hit

Access Pattern: A, C, E, G, A, C, I, K, A, C

Way1	Way2	Way3	Way4
1	1	0	0
A	C	I	K

Result: M, M, M, M, H, H, M, M, H, H

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

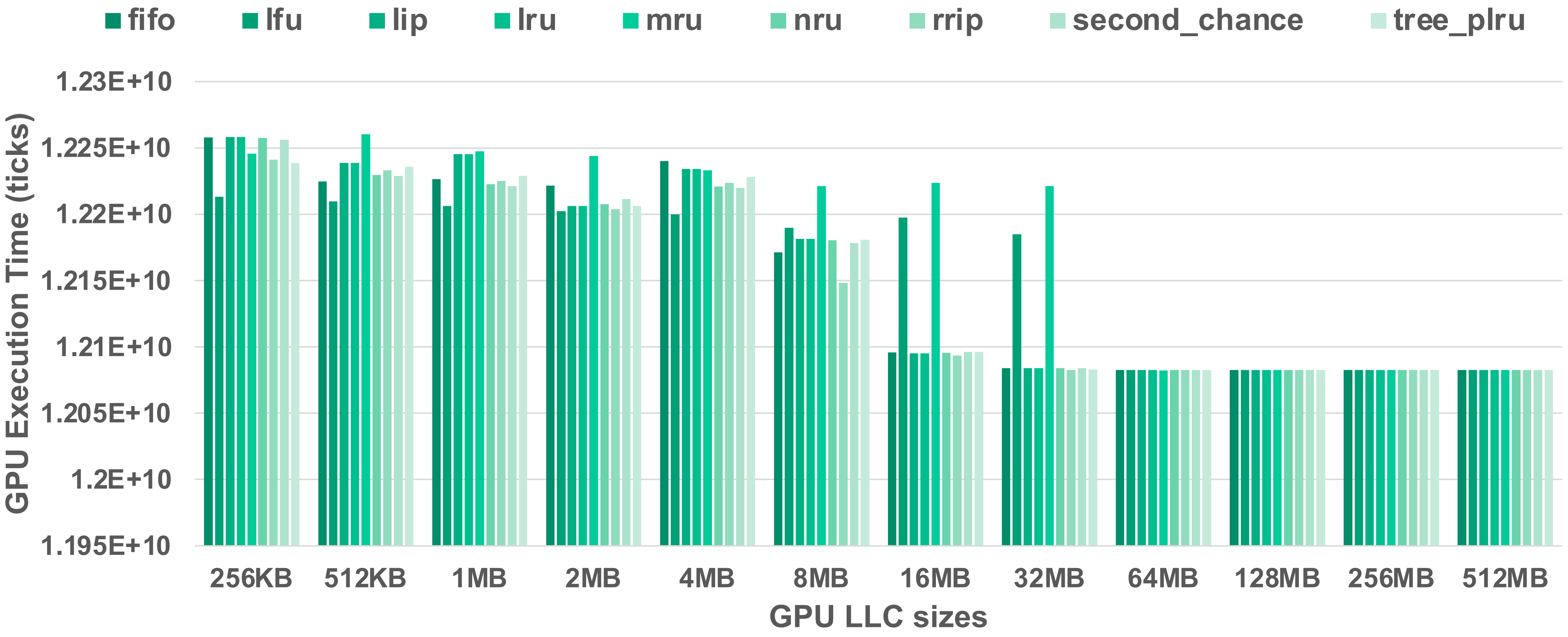
- System Setup:
  - Vega 20 GPU (**60** Compute Units, **16KB** L1 D$ per CU)
  - L1 latency: **143**, L2 latency: **260**, Scalar cache latency: **167**
    - Latencies based on Daniel & Vishnu's GAP work

- 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

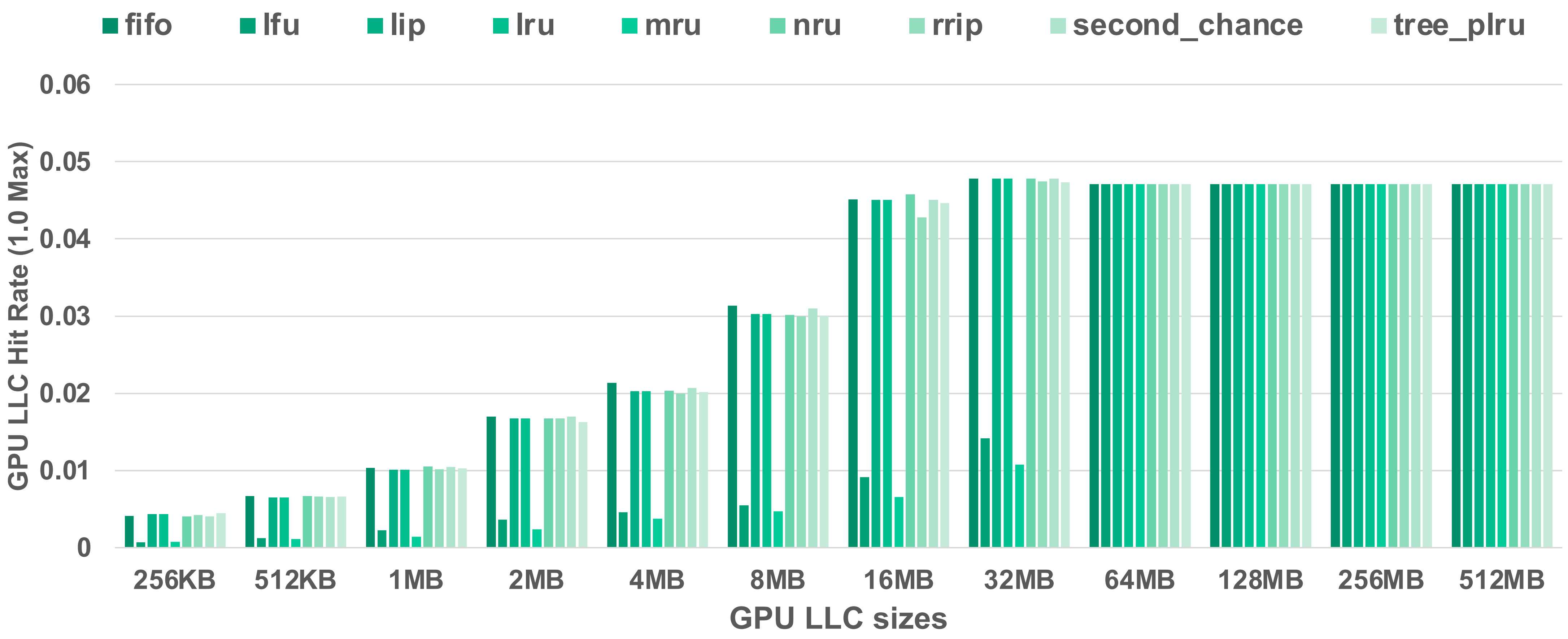
NW (Needleman–Wunsch) WT LLC Execution Time



LFU, MRU generally worse than others – Hurt temporal locality

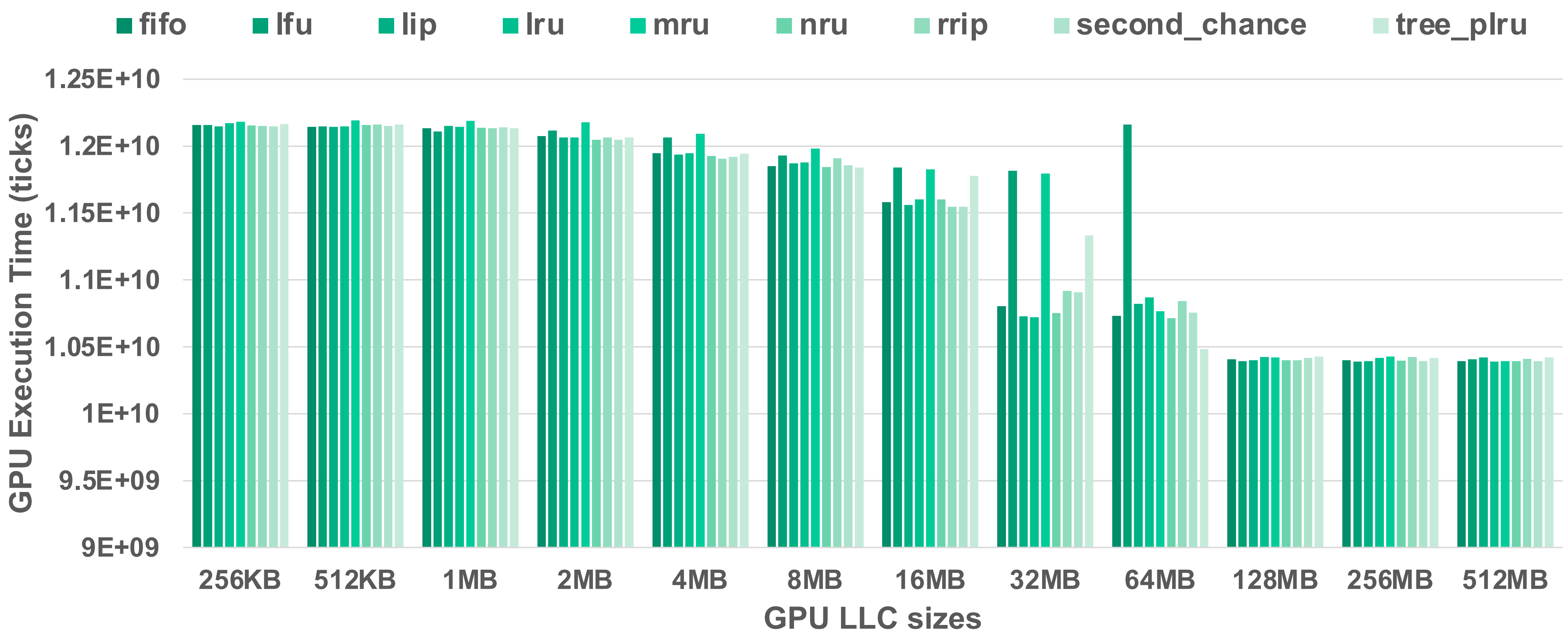
Little difference between rest of policies until WS fits in LLC

NW (Needleman–Wunsch) WT LLC Hit Rate



LLC Hit Rates confirm performance trends

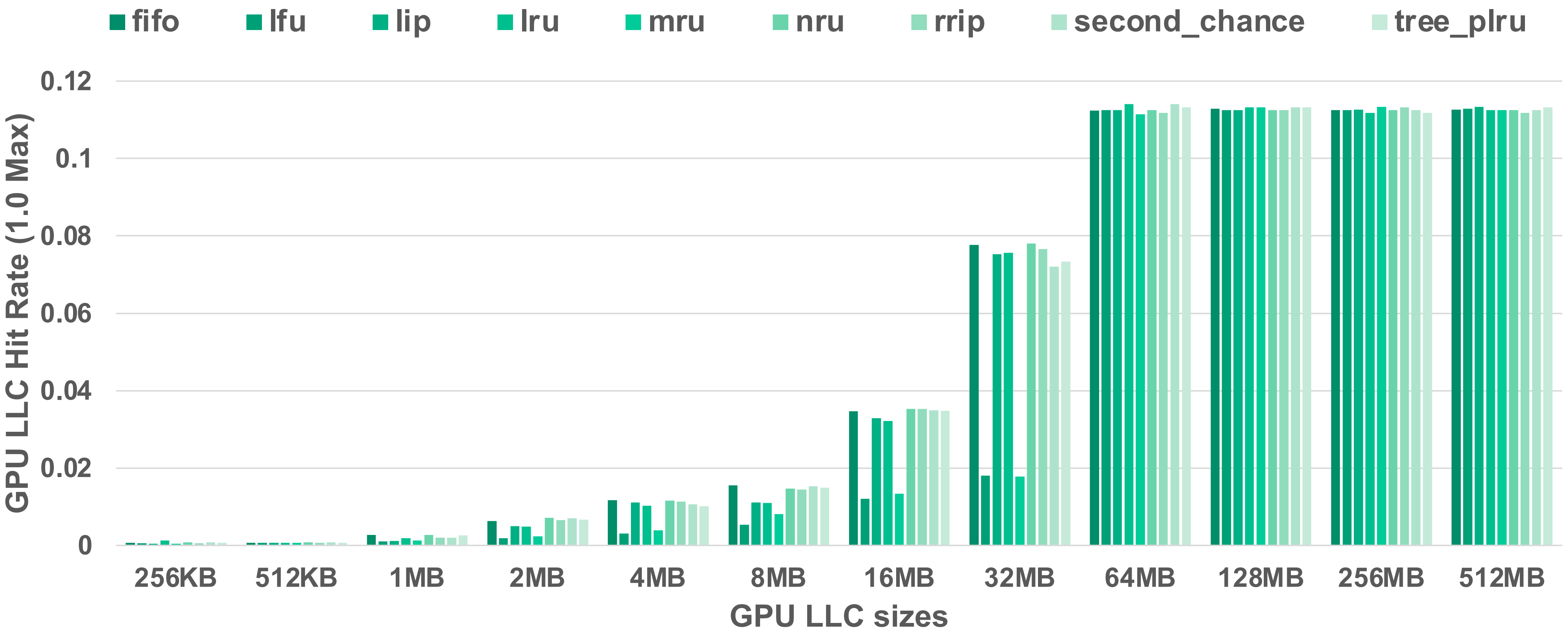
NW (Needleman–Wunsch) WB LLC Execution Time



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

Overall Result Takeaways

- 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