PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

VIRTUAL EVENT • MAY 24-25, 2022

# Enabling Memory Tiering in CacheLib

**Daniel Byrne**, Sergei Vinogradov, Igor Chorazewicz, and Tomasz Paszkowski

Intel Corporation

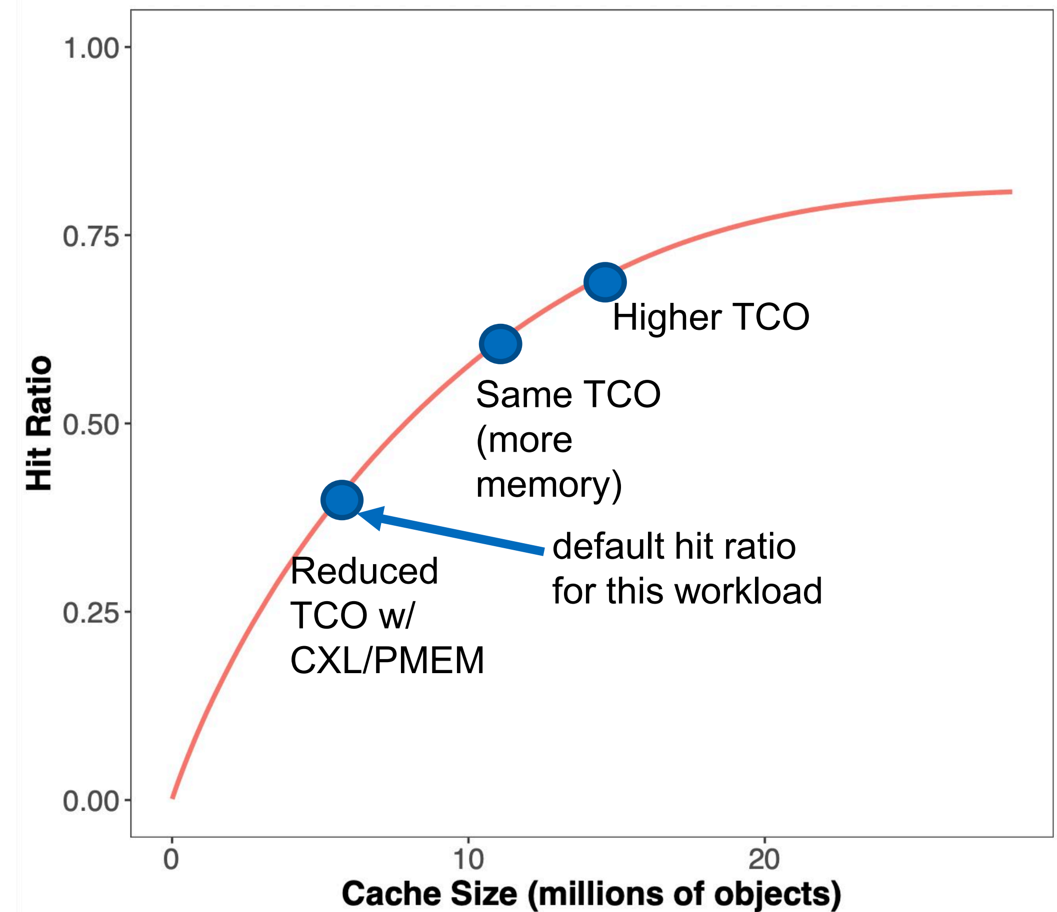
# Outline

- Why do we need memory tiering?
- Introduction to CacheLib
- Support for heterogenous memory in CacheLib
- Performance analysis
- Potential solutions to measured overhead
- Future directions
- Conclusion

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Why Memory Tiering?

- New memory technologies such as Compute Express Link (CXL) offer a wide range of use cases
  - Memory expansion in a single server
  - Dynamic memory pooling across nodes
- CXL can reduce cost of infrastructure
  - Increased memory utilization (dynamic memory pooling)
  - Reduced variants of server configurations
- Persistent memory offers more memory capacity at lower cost




 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + COMPUTATIONAL STORAGE SUMMIT 2022

# Introduction to CacheLib

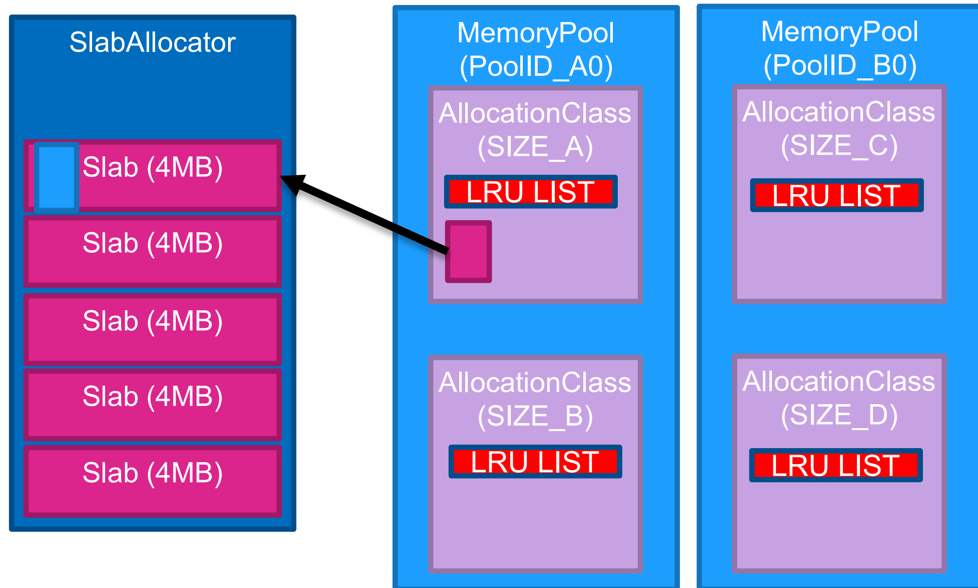
- CacheLib was developed by Meta and released as OSS in 2021
- CacheLib is a caching library – can run in-process or as a part of a network stack for remote access

```
auto item_handle = cache->find("key1");
if (item_handle) {
  auto data = reinterpret_cast<const char*>(item_handle->getMemory());
  std::cout << data << '\n’;
}

// getMemory can block if handle is not ready
auto handle = cache.find("foobar");
handle.onReady(processItem);
```

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Introduction to CacheLib – Memory Organization



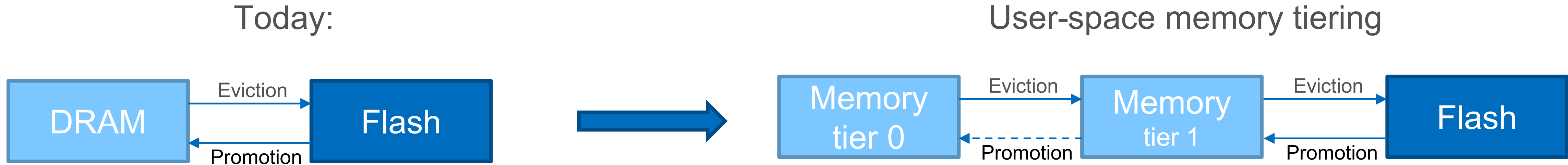

- Memory is divided into independent pools
- Each pool contains independent allocation classes
- Items are organized logically into allocation classes by their respective size in bytes
- Allocation classes follow a geometric sequence (96, 144, 216, 328 … )
- On eviction the LRU item for that class is removed

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + COMPUTATIONAL STORAGE SUMMIT 2022

# Heterogenous Memory Support

https://github.com/pmem/CacheLib




- What changed:
  - Config API was extended to allow configure memory tiers
  - No user-visible changes to the CacheLib applications
  - Re-use existing eviction mechanisms
  - Promotion among tiers is WIP

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

# Memory Tiers Configuration

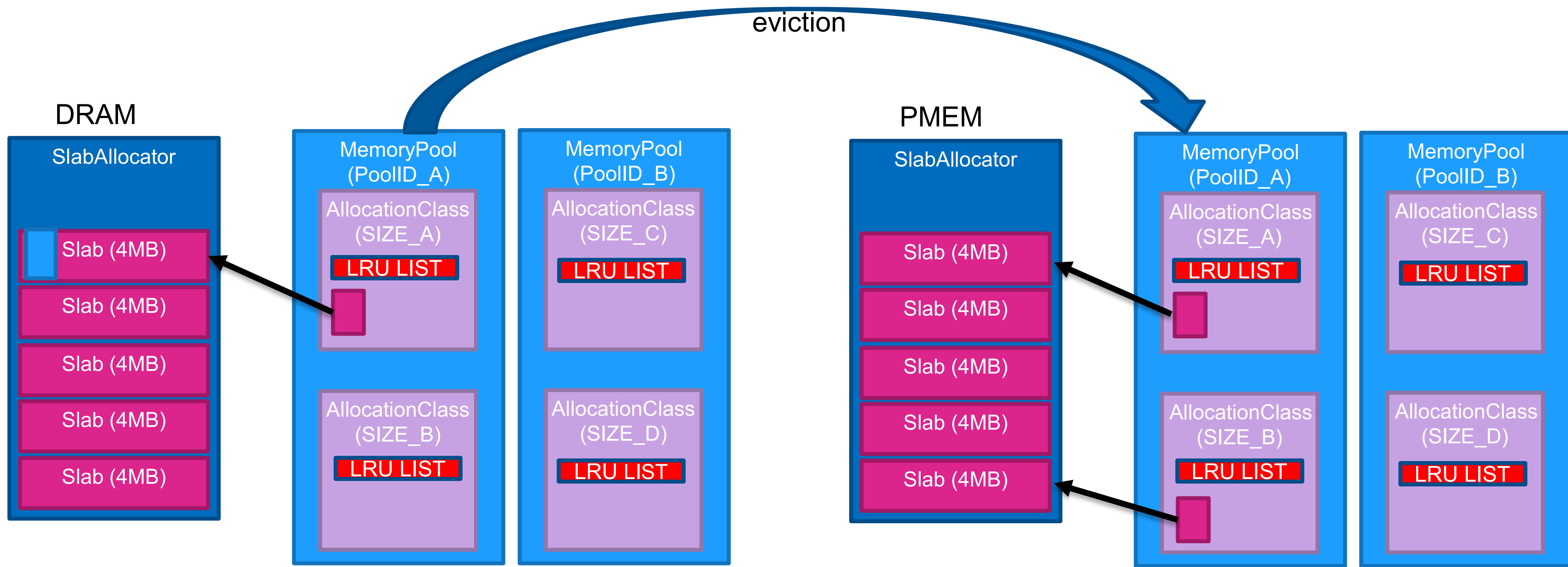
```
#include “cachelib/allocator/CacheAllocator.h”
std::unique_ptr<Cache> cache;
facebook::cachelib::PoolId default_pool;

void initializeCache() {
    Cache::Config config;
    config
        .setCacheSize(12 * 1024 * 1024 * 1024); // 12 GB – total size
        .setCacheName("My cache");
        // configure DRAM to have 4GB and PMEM to have 8GB
        .configureMemoryTiers( {
                MemoryTierCacheConfig::fromFile("/dev/shm/file1").setRatio(1),
                MemoryTierCacheConfig::fromFile("/mnt/pmem1/file1").setRatio(2)) })
        .validate();
    cache = std::make_unique<Cache>(config);
    default_pool = cache->addPool(
                        "default", cache->getCacheMemoryStats().cacheSize);
}
```

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + COMPUTATIONAL STORAGE SUMMIT 2022

# Multi-tier Memory Organization




- Memory pool and allocation classes are mirrored on each layer
- New elements are always inserted in topmost tier (if possible)
- Individual objects can be moved between tiers (evicted) on *insert()* or *find()* function
  - Requires locking LRU list in source and target tiers

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

# Benchmarks and system configuration

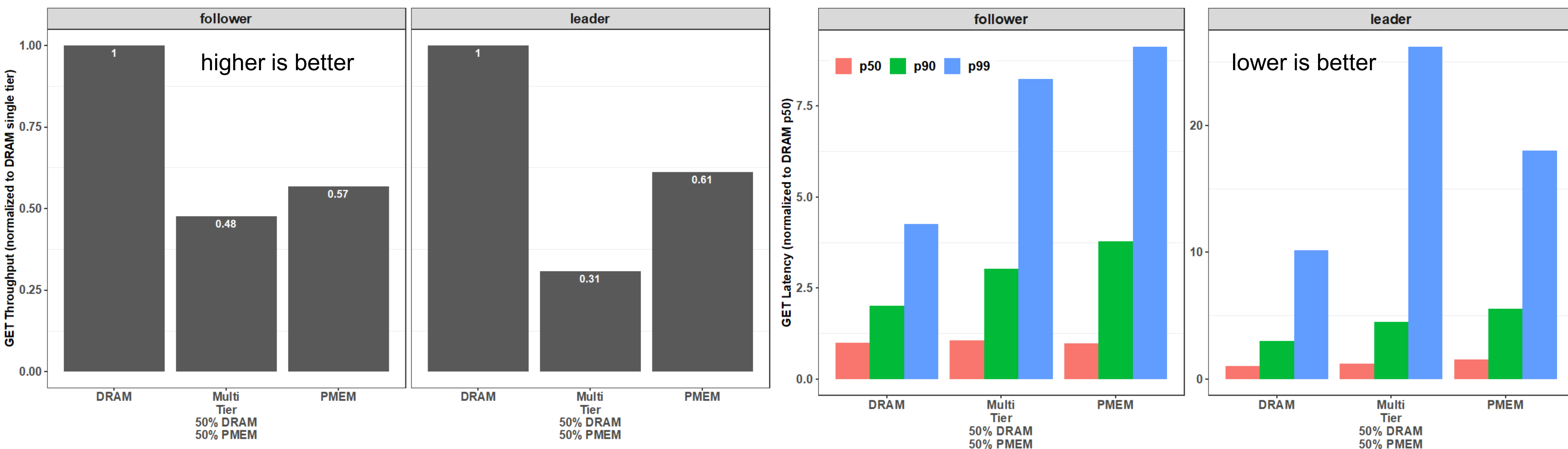
- 2x 3rd Gen Intel® Xeon ® processor (28core @ 2.60GHz)
  - 256 GB (16 slots/16GB/3200) total DDR4 memory
  - 2 TB (8x256GB) NMB1XBD256GQS Intel® Optane™ PMEM (Barlow Pass)
  - CentOS Linux release 8.5.2111, kernel v5.17.5
  - CacheLib compiled with gcc v8.5
  - Microcode 0xd000332 with HyperThreading and Turbo
  - Tested by Intel on 05/03/2022

- Benchmarks used from Cachebench:
  - Follower
    - https://github.com/byrnedj/CacheLib/blob/bg_and_tt_for_snia/cachelib/cachebench/test_configs/hit_ratio/graph_cache_follower_fbobj/config.json
  - Leader:
    - https://github.com/byrnedj/CacheLib/blob/bg_and_tt_for_snia/cachelib/cachebench/test_configs/hit_ratio/graph_cache_leader_fbobj/config.json
  - With memory tiers:
    - usePosixShm: true
    - //tier 1
      [ { ratio: 1, file:"/dev/shm/tier1" },
      //tier 2 (if enabled)
      { ratio: 1, file:"/mnt/pmem0/tier2" } ]

PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Initial Benchmarking Results for Lookaside Cache Workloads




- Graph Cache Leader - 0.42 hit ratio, 71.6M keys, 240M requests
- Graph Cache Follower – 0.91 hit ratio, 37.2M keys, 240M requests
- 8GB total cache size (4GB DRAM + 4GB PMEM for multi-tier)
- 24 requesting threads

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex .

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# VTune Performance Analysis

- Hotspot analysis report summary from VTune

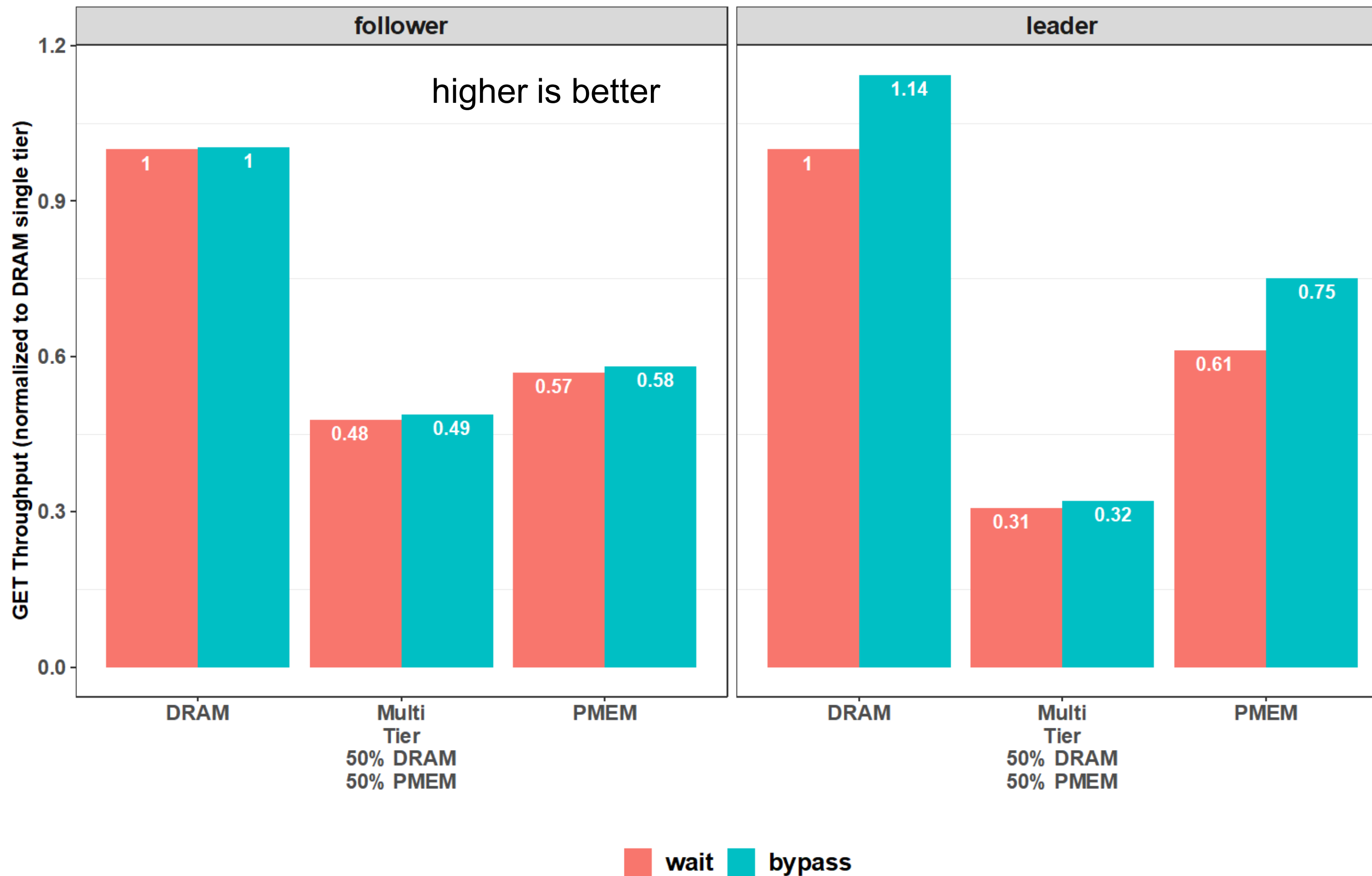
Grouping: Function / Call Stack

| Function / Call Stack | CPU Time ▼ |
|---|---|
| folly::detail::distributed_mutex::spin<folly::detail::distributed_mutex::Waiter<std::atomic>> | 36.404s |
| folly::hardware_timestamp | 28.096s |
| folly::detail::distributed_mutex::publish<folly::detail::distributed_mutex::Waiter<std::atomic>> | 13.632s |
| facebook::cachelib::RefcountWithFlags::getRaw | 5.950s |
| __memmove_avx_unaligned_erms | 4.111s |
| facebook::cachelib::CacheAllocator<facebook::cachelib::LruCacheTrait>::findEviction | 2.623s |
| std::__atomic_base<unsigned int>::operator&= | 2.277s |
| facebook::cachelib::CacheAllocator<facebook::cachelib::LruCacheTrait>::allocateInternalTier | 2.195s |
| facebook::cachelib::DListHook<facebook::cachelib::CacheItem<facebook::cachelib::LruCache | 2.018s |
| folly::detail::atomic_fetch_set_x86<unsigned long> | 1.991s |
| folly::detail::double_radix_sort_rec | 1.924s |
| facebook::cachelib::ChainedHashTable::Container<facebook::cachelib::CacheItem<facebook: | 1.849s |

LRU lock contention!

 | ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Testing the impact of always locking LRU queue




- **tryLockUpdate** – controls whether update LRU list on lookup
  - **wait**: always update
  - **bypass**: update only if there is no contention (std::try_to_lock)
- The problem is not multi-tier specific
  - DRAM-only & PMEM-only: find P99 latency improved by more than 2 times with **tryLockUpdate: true**

Performance varies by use, configuration and other factors. Learn more at [www.Intel.com/PerformanceIndex](www.Intel.com/PerformanceIndex) .

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Evictions: single-tier vs multi-tier

Evictions cost more in multi-tier environment.
Critical section under LRU lock:

**Eviction in the Single-Tier:**

- Find the victim and recycle it immediately.

**Eviction in the Multi-Tier:**

- Find the victim in the current tier
- Allocate new Item in the destination tier. Might cause recursive eviction.
- Copy data from the source tier to the destination one.
- Recycle Item from the source tier.

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

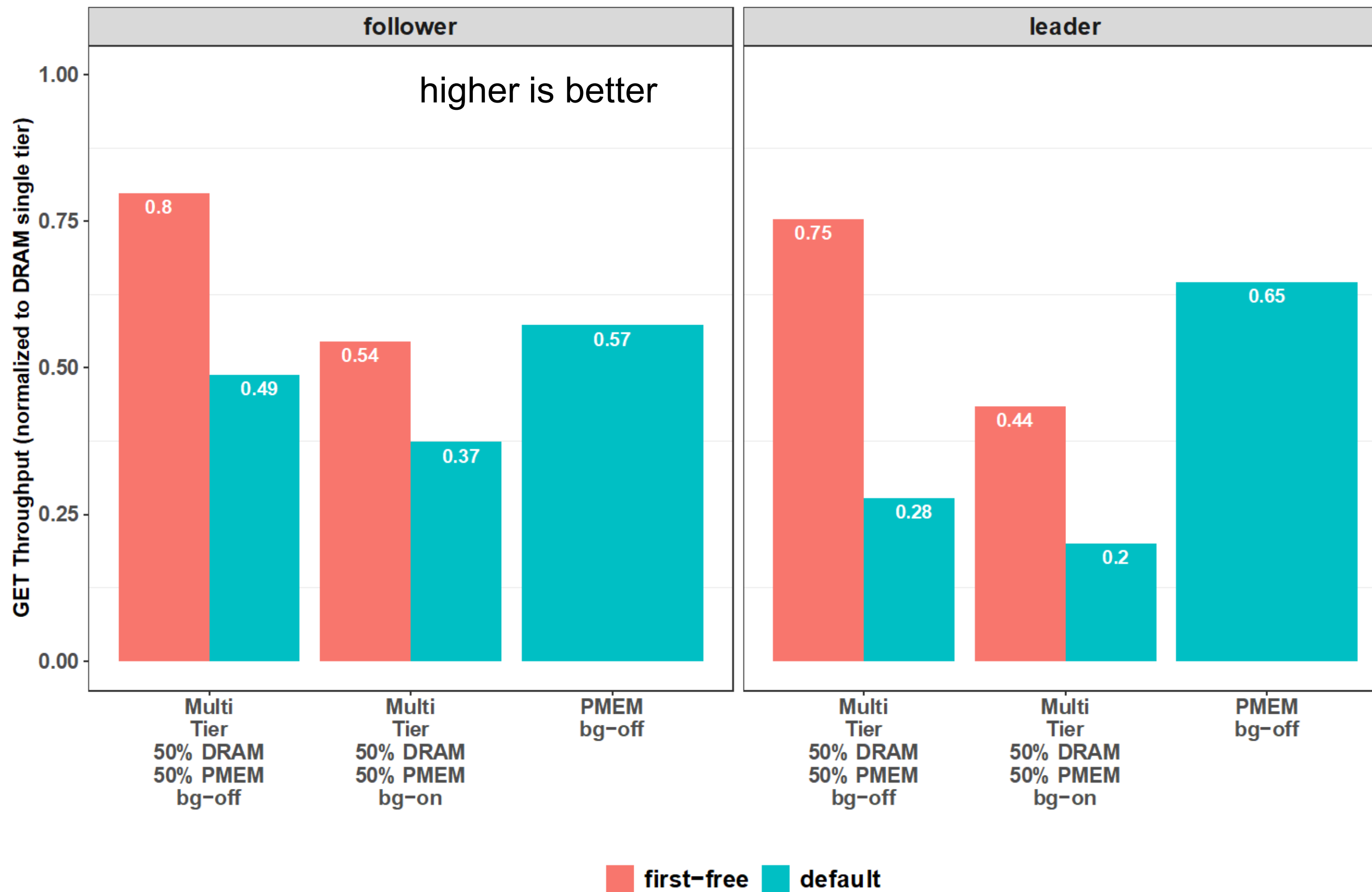
# Decrease Eviction Overhead - Possible Solutions

- Background eviction thread
- Insert to the first free tier
  - Ideally, used together with background thread
- Decrease critical section length
  - Remove the item from eviction queue drop the lock and evict
  - Evicting an object should be lock-free!
- Work in progress
  - Scalable eviction policy (LRU, LFU, etc.) implementation

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

# Insert to first free allocation slot & background eviction

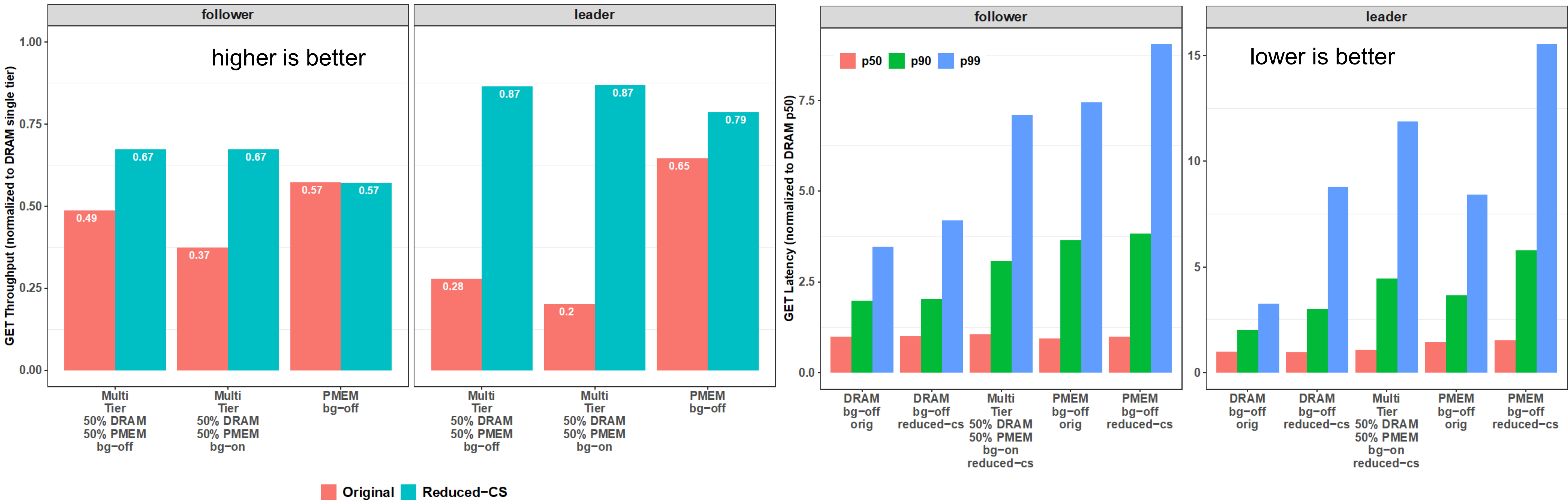



- First free for tiered instances: use first available allocation slot (not necessarily tier 0)
- Problem: Desired cache behavior may be to promote hot items from tier 1 to tier 0
  - Initial results show only about 20% success rate
  - Hot items are accessed often leaving a small chance that we acquire the locks needed to promote
- Background eviction: pre-emptively move cold items to the next memory tier
  - Done in batch
  - Can have negative effect since we add even more contention to the LRU locks under high throughput

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex .

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Decreasing Critical Section Length




- LRU lock for tier 0 is dropped once we have eviction candidate
- Multi-tier has higher throughput in both workloads & reduced latency over PMEM in follower workload

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex .

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

SNIA PERSISTENT MEMORY + COMPUTATIONAL STORAGE SUMMIT 2022

# Future Directions - Scaling Eviction Policy

- Container-less policies
  - Random sampling of objects in a class does not require maintaining LRU structure – similar to Redis and Hyperbolic Caching (ATC’17)
  - Need to add recency/frequency information per item
- Partitioned LRU lists – current memcached LRU
  - Cachelib 2Q implementation uses the same MultiList structure for {Hot,Warm,Cold} regions and requires locking entire structure for an operation
  - Use separate LRU lists for Hot, Warm, Cold to reduce contention

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Conclusion

- Memory tiering with CXL/PMEM increases memory capacity for a lower cost compared to DRAM
- Especially applicable to caching applications where more memory translates to higher hit ratio
- Software support needs to consider techniques to mitigate data movement overhead on the critical path
  - Choose fastest allocation path (first free method)
  - Background data migration
  - Reduced critical sections
- We believe we can get multi-tier DRAM+PMEM to performance parity with single tier DRAM instance

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.

# Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

No product or component can be absolutely secure.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

SNIA PERSISTENT MEMORY + COMPUTATIONAL STORAGE SUMMIT 2022

VIRTUAL EVENT • MAY 24-25, 2022

# Please take a moment to rate this session.

Your feedback is important to us.

 ©2022 Storage Networking Industry Association ©. Intel Corporation. All Rights Reserved.