Hazelcast & Memkind

hazelcast

Zoltán Baranyi

Senior Software Engineer
Hazelcast

intel

Michal Biesek

Software Engineer
Intel

Agenda

Memkind introduction

PMem why volatile in App Direct?

- Use large capacity
- Fine-grained control of memory placement
- Unified memory management for DRAM and PMem provided by memkind

Memkind overview

- Open-Source allocator, written in C - available on Linux

- Malloc-style API

```
void *memkind_malloc(memkind_t kind, size_t size);
void memkind_free(memkind_t kind, void* ptr);
…
```

- Memory (kind) detection mechanism

```
memkind_free(NULL, ptr)
memkind_t kind = memkind_detect_kind(ptr)
```

PMem in memkind

→ 2 operation modes

FS-DAX

- Kernel requirements >= 4.2
- Allocation based on a file - fallocate
- NUMA locality - fine-grain control
- Kind created by user
- Pool size management

KMEM-DAX

- Kernel requirements >= 5.1
- Allocation based on memory binding - mbind
- NUMA locality - limited control
- Kind created automatically
- COW support

About Hazelcast

About Hazelcast

- Distributed in-memory computing platform
- Infrastructure software
- K/V store, streaming, SQL, Entry Processors, …
- Big in-memory state
- Written in Java

Typical use-cases:

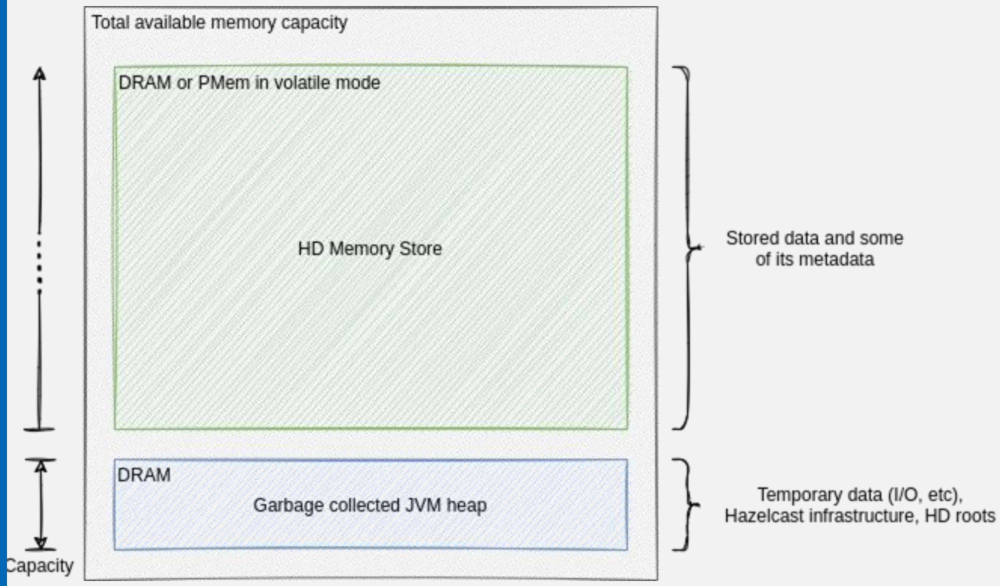
- Caching layer
- Web session
- Real-time analytics (Hazelcast Jet)
- Fraud detection
- Many more

Big heaps and the JVM

- Java heap is garbage collected
- GC algorithms
  - Traditional GCs: can't cope with big heaps or
  - Concurrent GCs: manage big heaps at the cost of CPU and memory overhead
- Hazelcast's solution: High-Density Memory Store

Hazelcast HD Memory Store

- Data outside of the Java Heap
- Not subject of garbage collection
- GC pauses don’t scale with the data
- No resource overhead (CPU, RAM)
- Memory is managed manually
- Persistent memory is a good fit
- More memory at a lower cost



Persistent memory in Hazelcast

- Used in Volatile App-Direct mode (Intel Optane PMem)
- Linux-only support – native code bundled in the jar

Hazelcast 4.0

- Allocation via libvmem
- Multi-socket support via LVM only
- NUMA-locality problems

Hazelcast 4.1

- Allocation via Memkind
- Native support to multi-socket machines
- Optional NUMA-awareness
- FS-DAX and KMEM-DAX modes supported

Integration with Memkind

Easy integration

- Hazelcast: malloc-like internal interface
- Memkind: malloc-like public interface

Hazelcast uses Memkind as page allocator with JNI calls

- The pages are thread-cached and split in the Java space
- Amortized JNI overhead cost

No uncommitting: pages are freed only at exit

- `MEMKIND_HOG_MEMORY` – no “hole punching” by `memkind_free()`

Convenient API for supporting various types of memory

- Whatever Memkind supports can be easily supported by Hazelcast
- PMem in FS-DAX or KMEM-DAX modes
- Potentially different NUMA policies
- DRAM with huge pages

Unified heap in FS-DAX mode

Challenge 1: Unifying the PMem heaps

- Each PMem mount point is a separate heap
- Having multiple heaps has to be transparent to the allocators
- Allocation strategy: which heap to allocate from?
  - Round-robin: balanced NUMA-Node utilization
  - NUMA-aware: all PMem accesses are NUMA-local

Challenge 2: Each block to be freed in its originating heap

- Tracking each allocation would be a massive overhead
- Memkind’s kind detection feature to the rescue

Hazelcast benchmarks

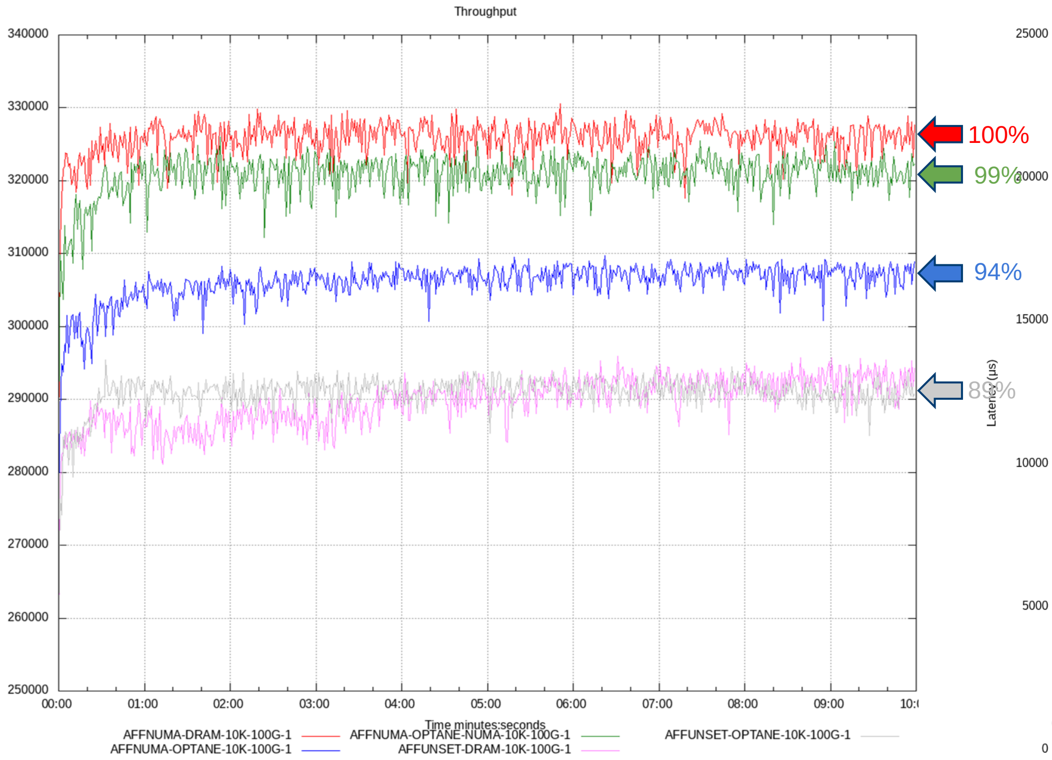
Caching benchmark

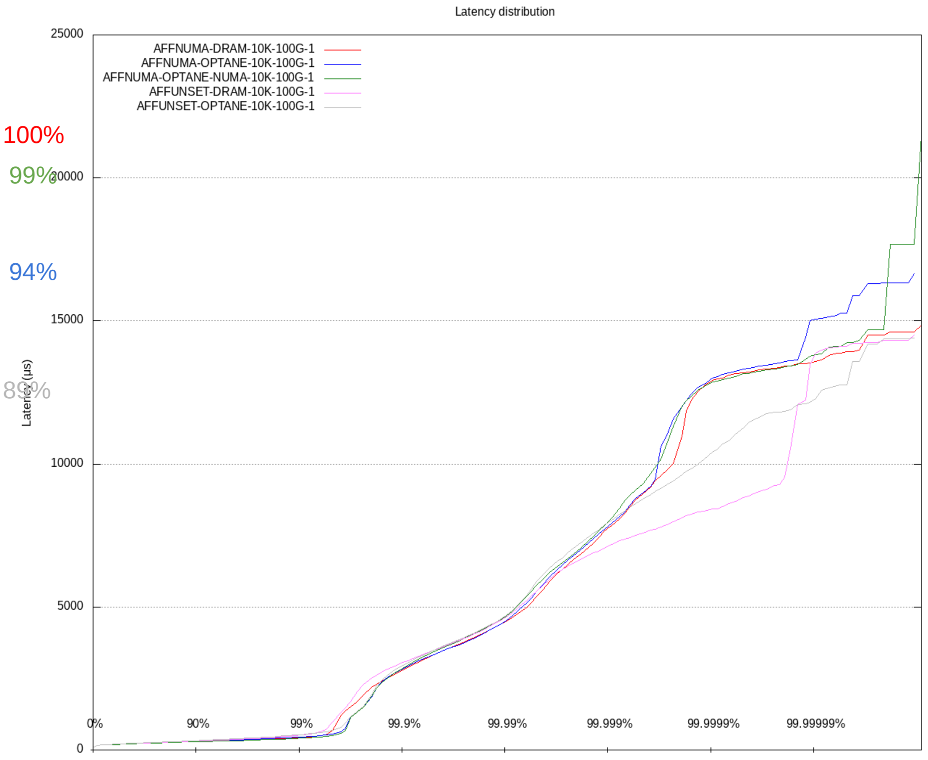
- **3 servers** on 3 dedicated machines
- Load from **20 clients** sharing 2 machines
- All on the same low-latency, 40Gbit/s network
- 50%-50% get-put
- **10KB entry size**
- **100GB** primary + 100 GB backup per server
- PMem in FS-DAX interleaved mode
- Medium and high load

Naming example:
`AFFNUMA-OPTANE-NUMA-10K-100G-1`

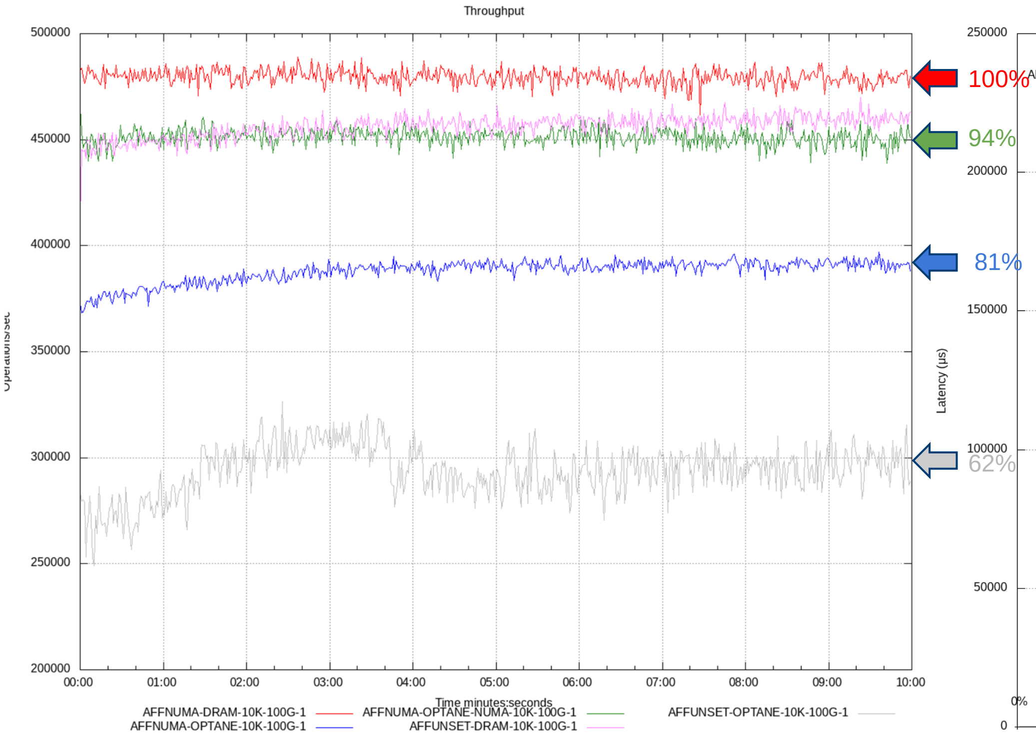
- **`AFFNUMA/AFFUNSET`**: whether the worker threads are NUMA-bound or not
- **`DRAM/OPTANE`**: HD memory store is backed by DRAM or Optane
- **`OPTANE-NUMA`**: Optane with NUMA-aware allocation policy
- **`10K`**: 10KB fixed entry size
- **`100G`**: primary data per member
- **`1`**: iteration of the test, always 1

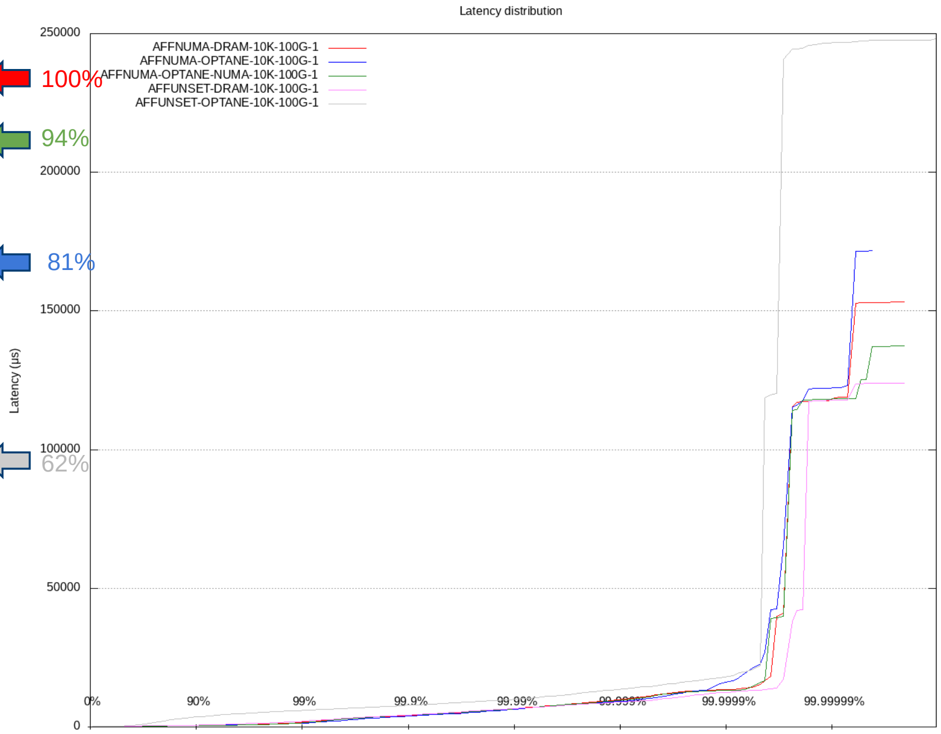
Caching benchmark – medium load





Caching benchmark – high load





Conclusion: NUMA-awareness matters

- NUMA-local PMem accesses may increase the performance by a lot
- The bigger the entry size and the load, the more significant the impact

Conclusion: PMem in distributed caching

- Networking can compensate for PMem's performance handicap
  - Serialization copying between buffers, inter-thread communication
- Traditional caching turned out to be a good use case
  - PMem used as a storage layer
  - Read-heavy workload
  - No or small performance penalty
- Load-intensive workloads still handicapped
  - Full scans
  - Hazelcast: Entry Processors
    - Data locality - processing entries selected by a predicate on the server

Summary

- Shown PMem can be in parity with DRAM in a distributed environment
- Shown how easy it is to integrate with Memkind
- Shown how Hazelcast brings PMem into the JVM ecosystem
- Blog post: pmem.io blog and Hazelcast blog
- Hazelcast manual: Using persistent memory

SPDK, PMDK, Intel® Performance Analyzers

Virtual Forum