1 Pmemkv 1.4 performance report

Testing date: 04/01/2021

1.1 Authors

Pawel Karczewski pawel.karczewski@intel.com

Igor Chorazewicz igor.chorazewicz@intel.com

Lukasz Stolarczuk lukasz.stolarczuk@intel.com

1.2 Audience and Purpose

This report is intended for people who are interested in evaluating pmemkv performance.

The purpose of the report is to compare performance of different engines which are described here and here.

1.3 Disclaimer

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.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

1.4 Test Setup

Testing date: 04/01/2021

Platform Hardware	
Manufacturer	Intel Corporation
Product	S2600WFT
Sockets	2
CPU Model	Intel(R) Xeon(R) Platinum 8280L CPU @ 2.70GHz
Cores per CPU	28
Total CPU Threads	112
DIMM Config	12 x 32 GB DDR4 2666 MT/s; 12 x 256GB Intel® Optane™ Persistent Memory, 2666MT/s
BIOS	SE5C620.86B.02.01.0013.121520200651
Hyper-threading	On
Turbo Boost	On

System Software	
OS	Ubuntu 20.04.2 LTS
Kernel	5.4.0-66-generic
gcc	9.3.0-17ubuntu1~20.04
Pmemkv	1.4-42-gf91d15c
Pmdk	1.10
Libpmemobj-cpp	1.12
TBB	2020.1-2

If not specified otherwise, benchmarks were run on fsdax mounted with `-o dax` flag. The file system was created on top of 6 interleaved 256GB Intel® Optane™ Persistent Memory DIMMs. Hereafter we will refer to a file on this filesystem as `$PMEM_FILE`. All benchmark processes were pinned to the closest numa node using numactl.

To learn how to provision Persistent Memory see here.

1.5 Introduction to pmemkv

pmemkv is a local/embedded key-value datastore optimized for persistent memory. Rather than being tied to a single language or backing implementation, pmemkv provides different options for language bindings and storage engines.

For more information, including **C API** and **C++ API** see: https://pmem.io/pmemkv. Documentation is available for every branch/release. For most recent always see (**master** branch):

- C++ docs,
- C manpage libpmemkv(3).

pmemkv is written in C/C++ and can be used in other languages - Java, Node.js, Python, and Ruby.

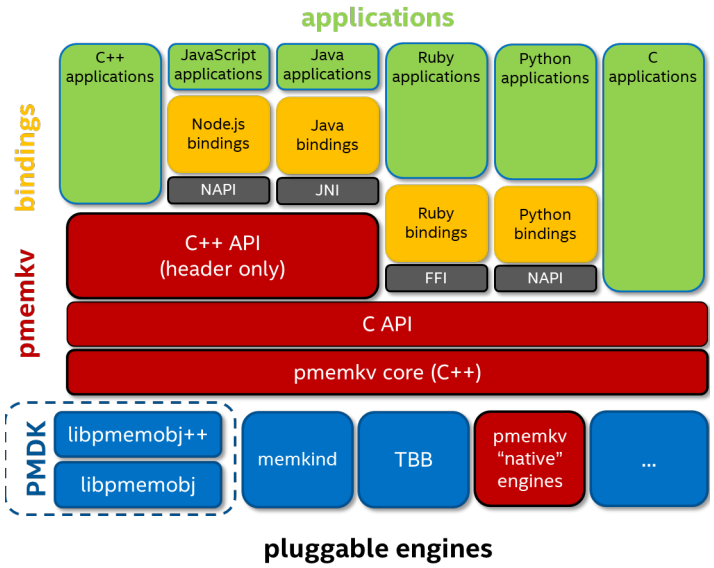

Figure 1: Pmemkv architecture

pmemkv offers several different storage engines which are tuned for different use cases. For list of all engines, see:

- always up-to-date README @ github.com/pmem/pmemkv

1.6 List of engines compared in this report

Some of the engines are marked 'experimental'. This means that they are not production quality yet and their behavior might change.

1.6.1 cmap

A persistent concurrent engine, backed by a hashmap that allows calling get, put, and remove concurrently from multiple threads and ensures good scalability. Rest of the methods (e.g. range query methods) are not thread-safe and should not be called from more than one thread. Data stored using this engine is persistent and guaranteed to be consistent in case of any kind of interruption (crash / power loss / etc).

1.6.2 csmap (experimental)

A persistent, concurrent and sorted engine, backed by a skip list. All methods of csmap are thread safe. Put, get, count_* and get_* scale with the number of threads. Remove method is currently implemented to take a global lock - it blocks all other threads. Data stored using this engine is persistent and guaranteed to be consistent in case of any kind of interruption (crash / power loss / etc).

1.6.3 radix (experimental)

A persistent, sorted (without custom comparator support), single-threaded engine, backed by a radix tree. For single-threaded workloads it offers the best bandwidth and latency from all sorted engines. Data stored using this engine is persistent and guaranteed to be consistent in case of any kind of interruption (crash / power loss / etc).

1.6.4 stree (experimental)

A persistent, single-threaded and sorted engine, backed by a B+ tree. Data stored using this engine is persistent and guaranteed to be consistent in case of any kind of interruption (crash / power loss / etc).

1.6.5 robinhood (experimental)

A persistent and concurrent engine, backed by a hash table with Robin Hood hashing. It uses only fixed size keys and values (8 bytes). Data stored using this engine is persistent and guaranteed to be consistent in case of any kind of interruption (crash / power loss / etc).

For people interested in comparing pmemkv engines to DRAM-based data structures pmemkv implements **dram_vcmap** engine, which is based on tbb::concurrent_hash_map. It is not covered in this report.

1.7 Microbenchmarks

Microbenchmarks were performed using pmemkv-bench. pmemkv-bench is a tool based on leveldb's db_bench and offers similar functionalities.

Following command template was used to run the benchmarks:

```
numactl --cpubind=file:$PMEM_FILE ./pmemkv_bench --db=$PMEM_FILE --num=10000000
--engine=$ENGINE --benchmarks=$BENCHMARKS --key_size=8 --value_size=$VALUE_SIZE
--threads=$THREADS --histogram=1 --db_size_in_gb=200
```

In this report following scenarios were tested:

- fillseq - inserting N values in sequential key order
- fillrandom - inserting N values in random key order

- readseq - read N values in sequential key order
- readrandom - read N value in random key order
- readwhilewriting - 1 thread doing random writes, T threads doing random reads
- readrandomwriterandom - T threads performing random read and write operations (each thread performs reads and writes alternately in batches, e.g. 90 reads followed by 10 writes), ratio of reads to writes is 9:1

N means number of elements to process and T means number of threads.

If multiple threads are used, each thread performs operations on all keys. This means that the total number of operations is `N * number of threads`. For sequential benchmarks, each thread operates on the same elements, in the same order.

In all random benchmarks, keys are generated accordingly to uniform distribution.

All of the following benchmarks were run with `N = 10 000 000`.

All of the following benchmarks were run with `key_size = 8 Bytes` and `value_size = 8 Bytes` (unless specified otherwise).

List of all figures is provided at the end of the document.

1.8 Multi-Threaded (MT) engines comparison - threads vs ops/sec

This section compares the performance of cmap, csmap, and robinhood for multi-threaded workloads. Performance results for readrandom, readseq, readrandomwriterandom, and readwhilewriting are presented twice - in figures 4,5,6, and 7 without robinhood and in figures 8,9,10, and 11 with robinhood. This was done to improve readability since the throughput difference between robinhood and cmap/csmap is large.

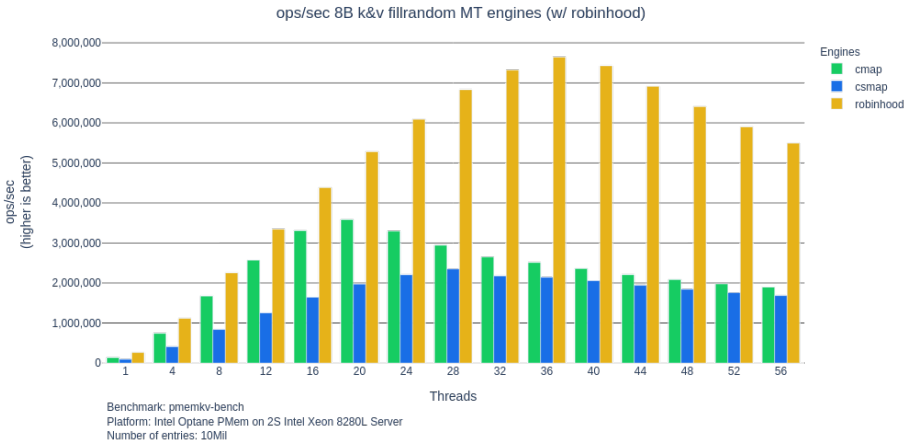

Figure 2: ops/sec 8B k&v fillrandom MT engines (w/ robinhood)

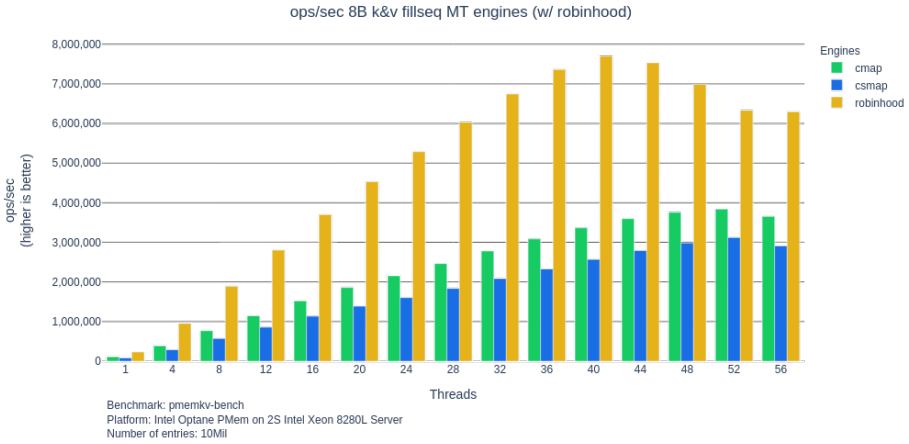

Figure 3: ops/sec 8B k&v fillseq MT engines (w/ robinhood)

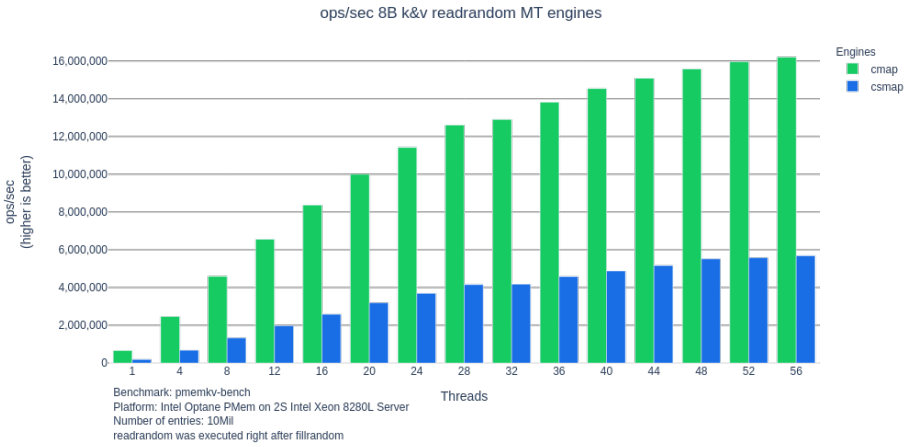

Figure 4: ops/sec 8B k&v readrandom (after fillrandom) MT engines

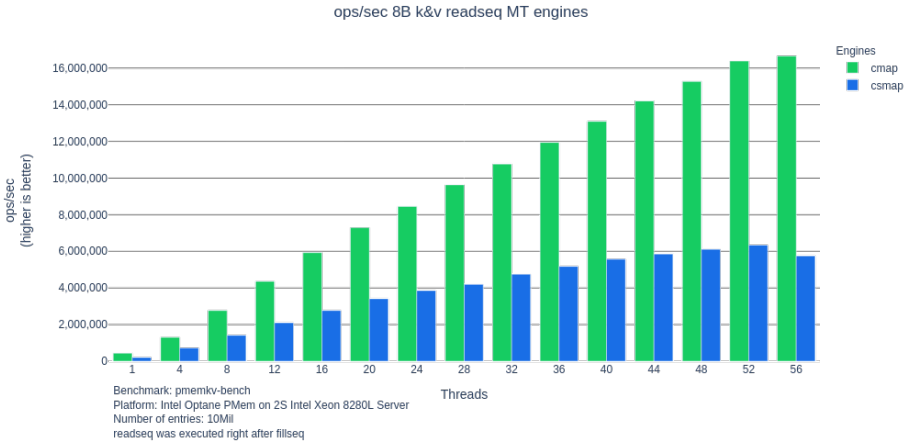

Figure 5: ops/sec 8B k&v readseq (after fillseq) MT engines

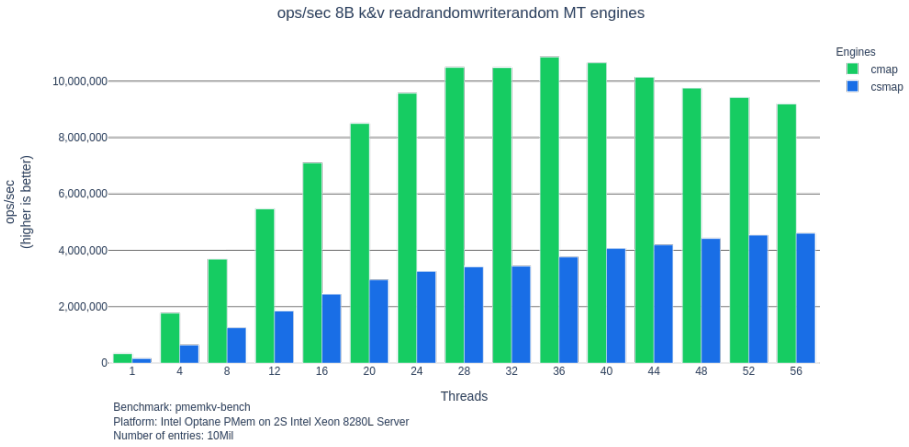

Figure 6: ops/sec 8B k&v readrandomwriterandom MT engines

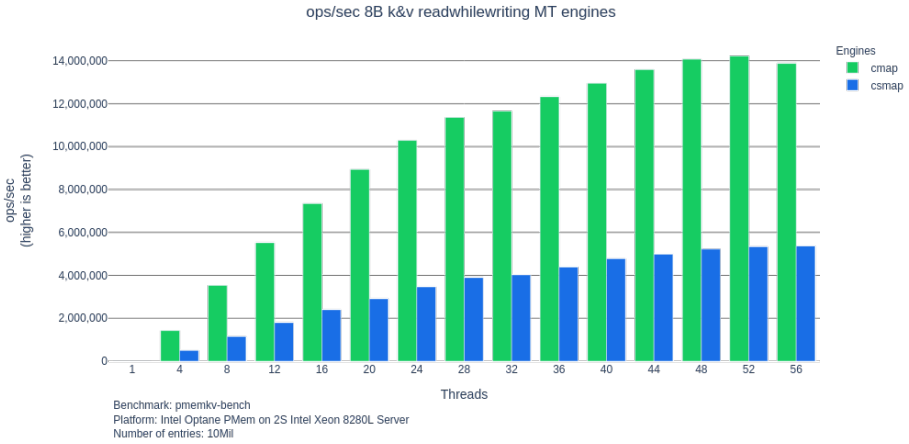

Figure 7: ops/sec 8B k&v readwhilewriting MT engines

Figure 8: ops/sec 8B k&v readrandom (after fillrandom) MT engines (w/ robinhood)

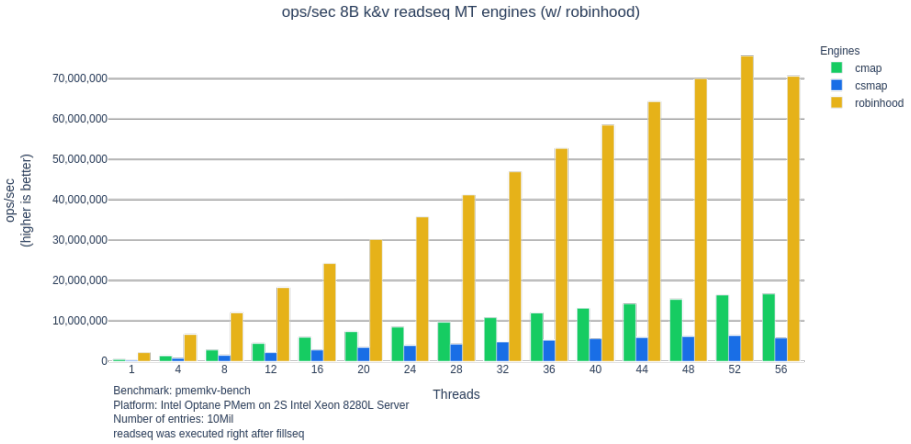

Figure 9: ops/sec 8B k&v readseq (after fillseq) MT engines (w/ robinhood)

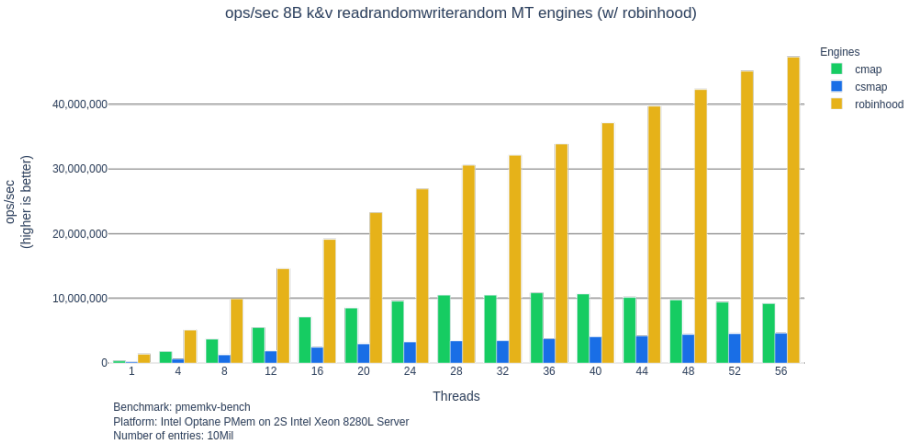

Figure 10: ops/sec 8B k&v readrandomwriterandom MT engines (w/ robinhood)

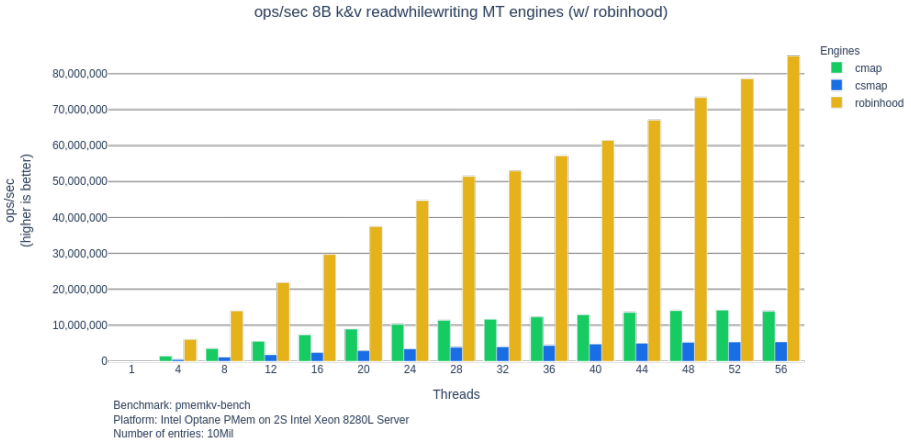

Figure 11: ops/sec 8B k&v readwhilewriting MT engines (w/ robinhood)

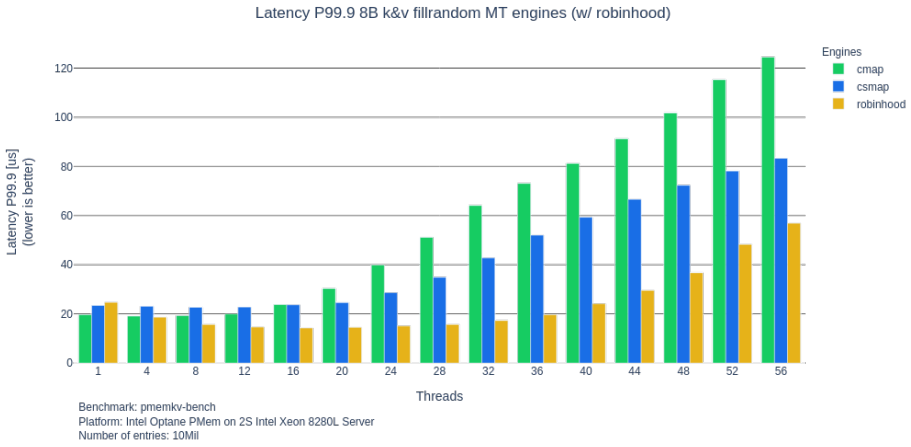

Figure 12: Latency P99.9 8B k&v fillrandom MT engines (w/ robinhood)

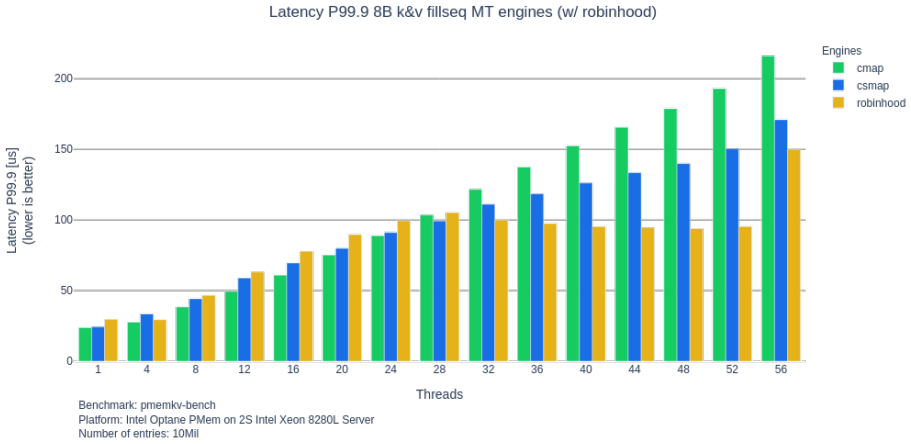

Figure 13: Latency P99.9 8B k&v fillseq MT engines (w/ robinhood)

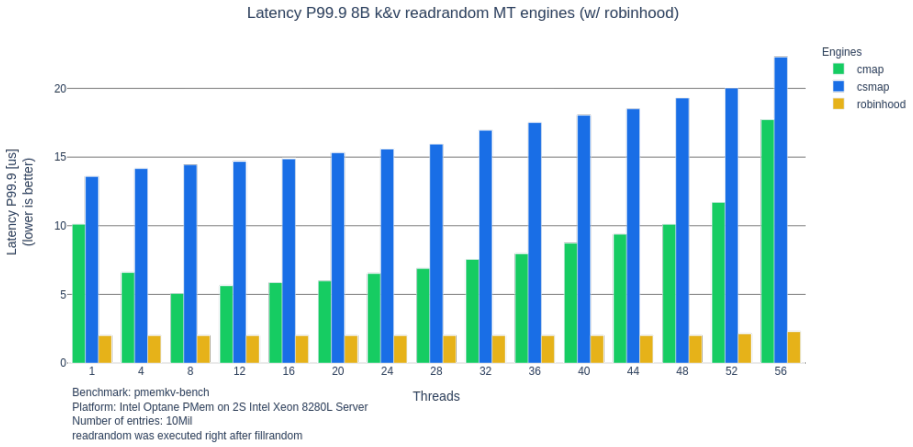

Figure 14: Latency P99.9 8B k&v readrandom (after fillrandom) MT engines (w/ robinhood)

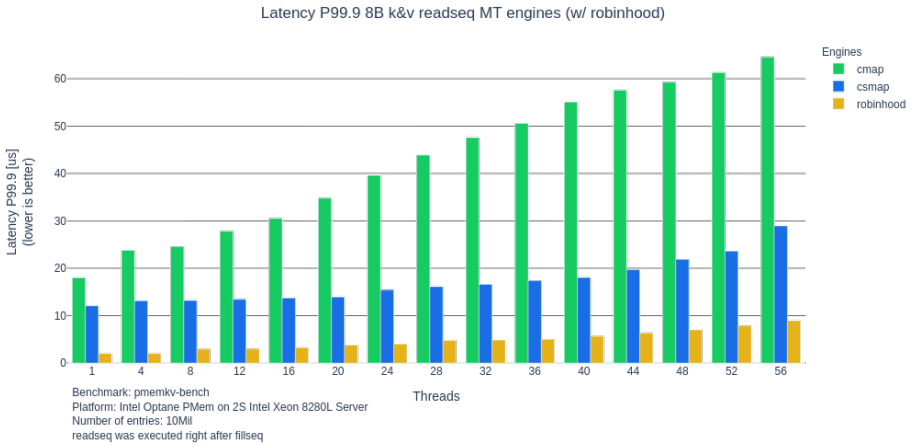

Figure 15: Latency P99.9 8B k&v readseq (after fillseq) MT engines (w/ robinhood)

1.8.1 Summary

In all benchmarks, robinhood shows the best performance. For write benchmarks, it scales up to 36 threads whereas cmap scales only up to 20 threads and csmap up to 24. For read benchmarks, robinhood scales up to the number of available threads and offers an order of magnitude better performance.

The difference in scaling comes mainly from the fact that operations on robinhood engine result in less cache misses (due to the data structure layout) and that robinhood locks are stored in DRAM, whereas cmap and csmap locks are stored in PMEM. It's important to keep in mind that robinhood only supports 8 byte keys and values.

1.9 Single Thread (ST) benchmarks' execution - value size vs ops/sec

Comparison of ops/sec results when value of different sizes is used. All engines run on a single thread.

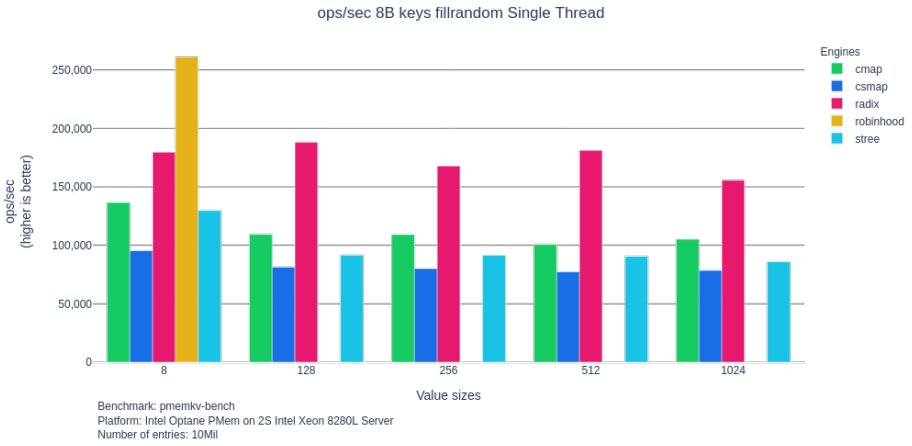

Figure 16: ops/sec 8B keys fillrandom Single Thread

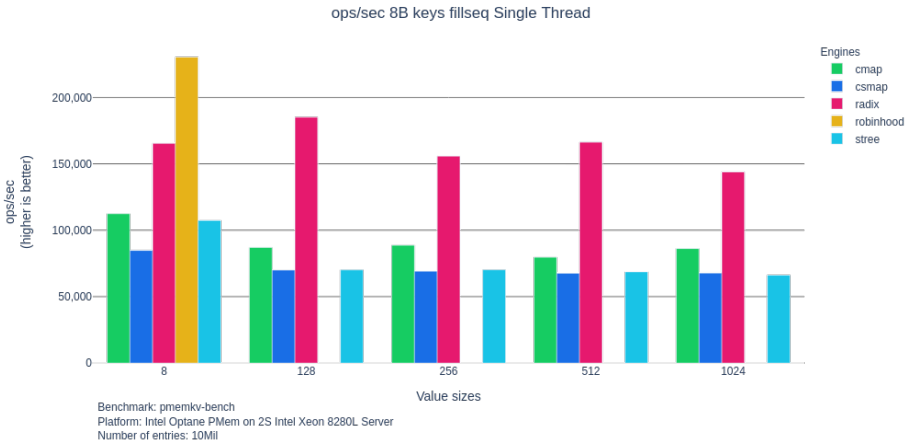

Figure 17: ops/sec 8B keys fillseq Single Thread

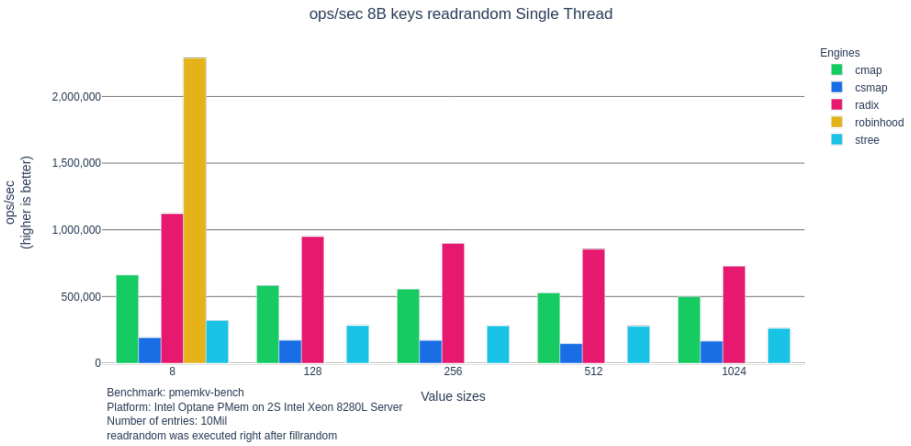

Figure 18: ops/sec 8B keys readrandom (after fillrandom) Single Thread

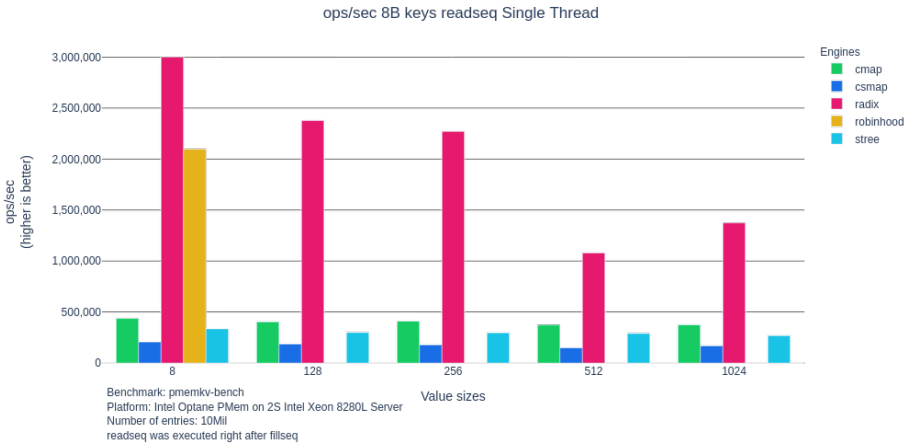

Figure 19: ops/sec 8B keys readseq (after fillseq) Single Thread

1.9.1 Summary

The best performance is achieved by robinhood and radix engine. The biggest difference can be seen in readseq benchmark where radix is over 6 times better than cmap and csmap. The reason for this difference is that radix layout is optimized for sequential access and that data is stored inline. Reads result in less cache misses than for cmap and csmap. In radix engine the allocation pattern is also more optimal for sequential access.

1.10 Multi-Threaded benchmarks - value size vs ops/sec

Additional comparison of value sizes for multiple threads.

1.10.1 cmap

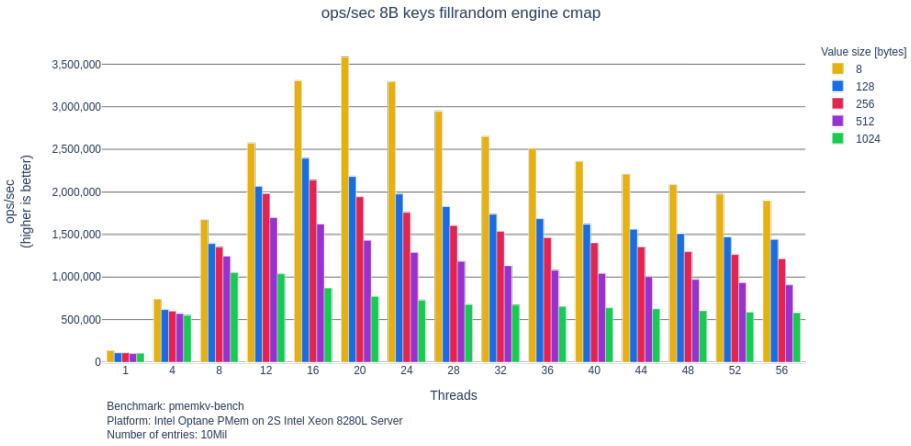

Figure 20: ops/sec 8B keys fillrandom engine cmap

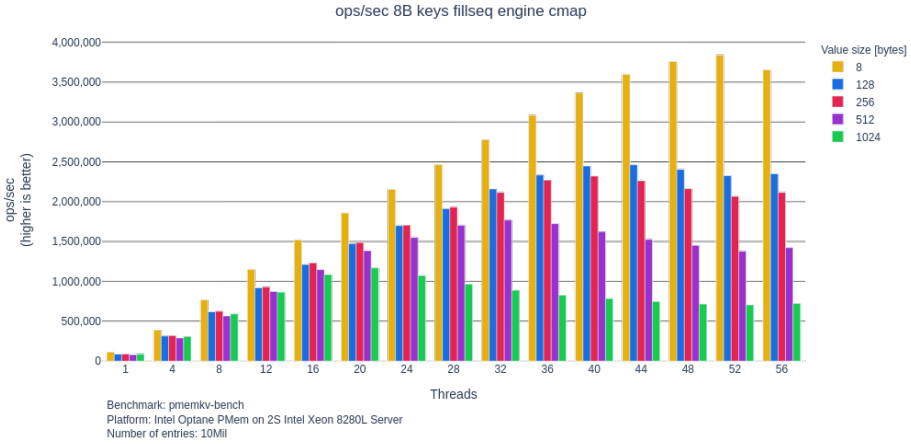

Figure 21: ops/sec 8B keys fillseq engine cmap

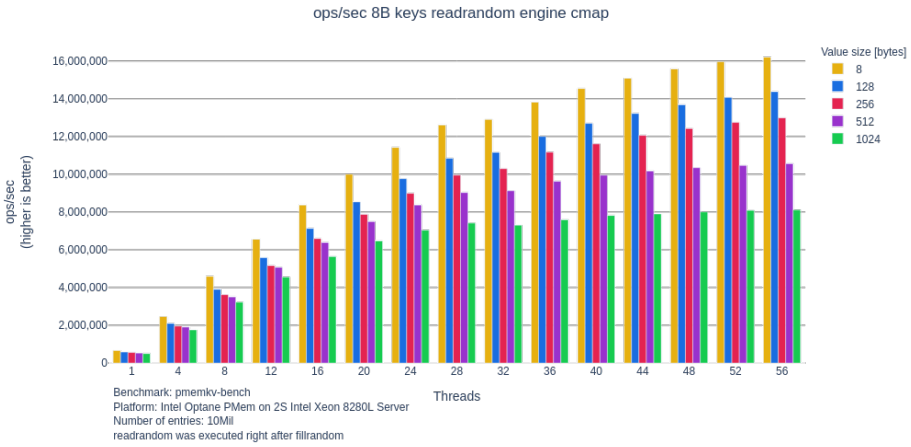

Figure 22: ops/sec 8B keys readrandom (after fillrandom) engine cmap

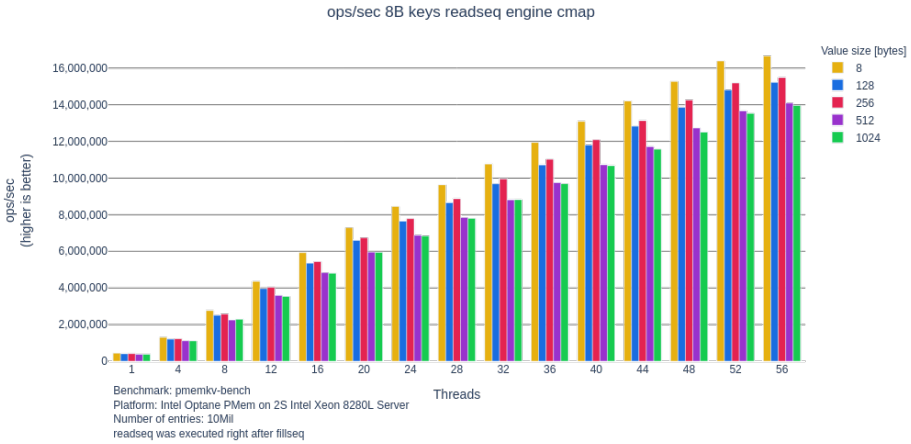

Figure 23: ops/sec 8B keys readseq (after fillseq) engine cmap

1.10.2 csmap

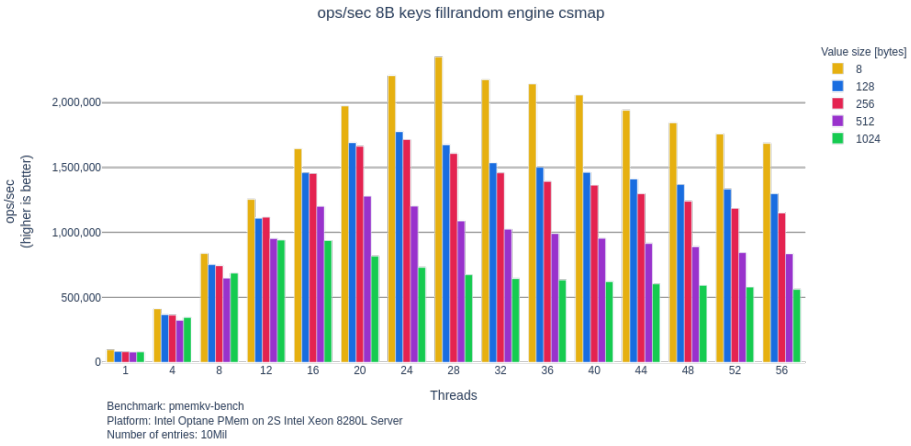

Figure 24: ops/sec 8B keys fillrandom engine csmap

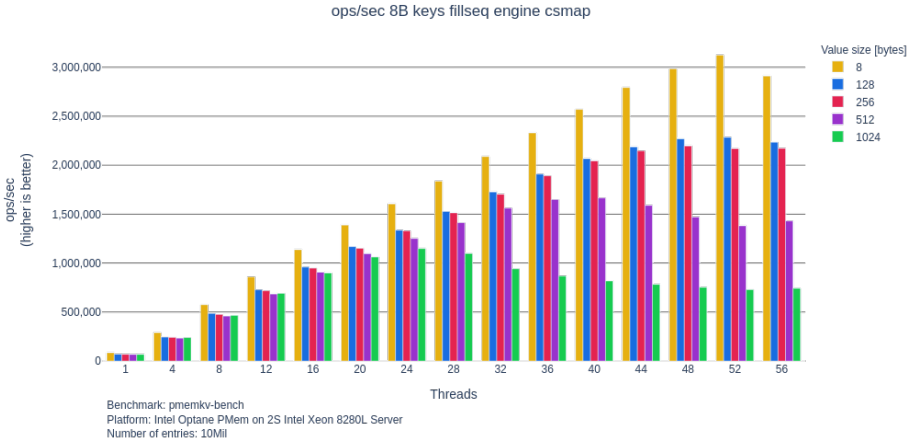

Figure 25: ops/sec 8B keys fillseq engine csmap

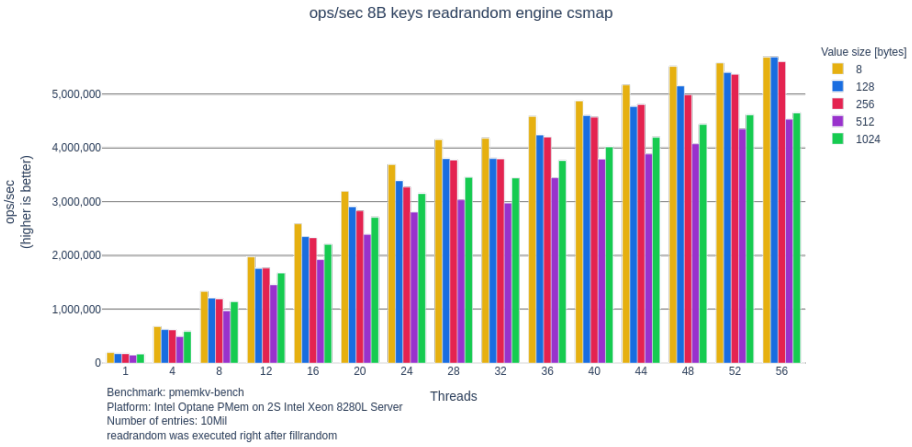

Figure 26: ops/sec 8B keys readrandom (after fillrandom) engine csmap

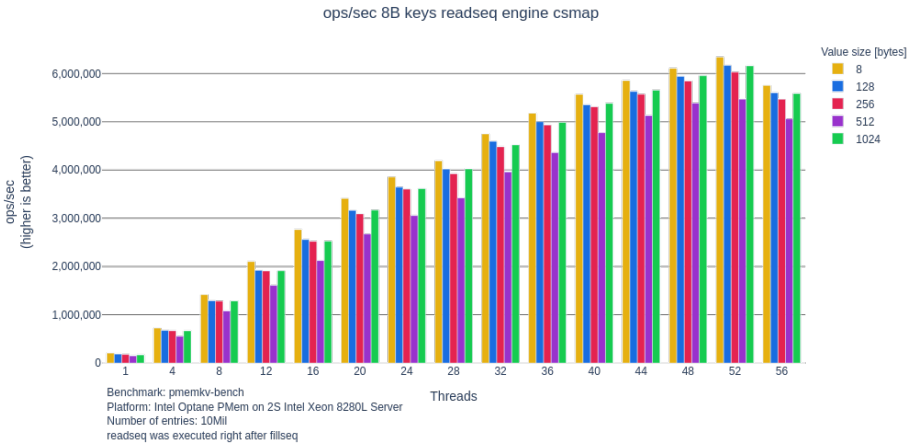

Figure 27: ops/sec 8B keys readseq (after fillseq) engine csmap

1.10.3 Summary

Comparing the two engines, cmap performs better in most cases for selected value sizes. For sequential benchmarks (especially for reads-only in sorted engine: csmap's readseq) all elements are accessed in the same order, which results in a high contention. This hides the most of differences in performance for different value sizes. With no surprise 8 bytes values provides the highest number of operations/second in all cases. The difference in results is mostly observed

in writes-only benchmarks, where a value size influences the ops/sec metric the most. As for reads-only workloads, the only "exception" in pattern is cmap's readrandom workload - results (instead of being close to each other) are reduced around 2 times for 1024 bytes (in comparison to 8 and 128 bytes values).

List of Figures