McGill University
School of Computer Science

Introducing Persistent Memory to LSM-tree based Key-Value stores

Namdar Kabolinejad
Supervisor: Prof. Oana Balmau

ABSTRACT

Key-value stores' high performance and space efficiency have turned them into a critical component in modern computing. To improve performance key-value stores take up large amounts of DRAM to cache highly accessed data. However, given the high cost and power consumption of DRAM and the demand for it by many data-intensive applications, it is crucial to be mindful of how the limited space of DRAM is being used. Novel byte-addressable permanent memory (PMEM) technologies offer data persistence at close-to-DRAM speed but at a much less cost per gigabyte, making them a promising complement to DRAMs. In this paper, I use a common KV-store type, the LSM-tree, and study the placement of different LSM components, specifically block caches and memtables, on DRAM and PMEM. I found that placing the entire block cache on PMEM increases the latency of the application by 15% on average, but alleviates DRAM space in the range of tens of MBs.

1 INTRODUCTION

Key-value stores (KVS) have turned into a critical component in modern computing. They are commonly used in storage engines to support data-intensive applications and achieve high performance and space efficiency. Due to their flexibility, ease of use, and efficiency KVS are quickly gaining more popularity between users and developers.

There are two common types of KVS, one based on Log-Structured Merge Trees (LSM-tree) and the other based on B-trees (a generalization of a binary search tree) [1]. The LSM tree-based KVS are designed to efficiently support write-intensive workloads and are used in databases such as BigTable, LevelDB, RocksDB.

For persistence, the data of KVS are stored to devices with a block-based interface, however, to increase read and write performance DRAM is used as an extra layer of memory - such that highly accessed key-value pairs (hot data) are stored in DRAM.

Given the rising costs of DRAM, it is crucial to be mindful of how we are using the limited amount of main memory in our machines. By reducing the amount of workload on DRAMs or decreasing the amount of DRAM used we will be able to run larger data sets with higher performance or run additional programs on our machine.

Novel byte-addressable permanent memory (PMEM) technologies offer data persistence at close-to-DRAM speed and are less expensive than DRAM to acquire, making them a promising complement to DRAMs. High-capacity non-volatile memory (NVMs) which are a tier of PMEM, can increase the total memory capacity of a server by up to 8 times while costing much less per gigabyte [2]. Therefore, leveraging PMEMs in computing has a large potential to improve performance at a low cost, however, this is only if it is used properly and managed well. Naturally, the question of whether it is possible to use PMEM instead of DRAM in KV-stores without loss of performance arises.

In recent work, a group of researchers created HeMem [2], a tiered main memory management system designed from scratch for commercially available NVM with a goal to optimize the tiered DRAM+NVM servers. Furthermore, there has been some work done on creating a new high-performance persistent range following the PAC guidelines and PMem-specific access patterns to improve the performance of NVMs as well [3]. Both these researchers provide new software and algorithms to

optimize the performance of NVMs in systems and work at an OS level. However, in this research, I am looking into using PMEM directly in KVS without the overhead of OS.

The goal is to take an LSM-tree based KVS and study the placement of the different LSM components. I specifically experiment with allocating block caches and memtables of the LSM-tree based KVS, on DRAM or PMEM while maintaining or improving the default performance-having everything in the main memory.

I will be running experiments on RocksDB, a popular LSM-tree backed database developed by Facebook. I will be trying to place the block cache and memtable of the database on different memory tiers and measure the performance of the database for each scenario with multiple workloads.

Given the potential of PMEMs and the significance of KV-stores, it would be advantageous to find a way to place KVS data structures in PMEM without loss of performance. The solution can be implemented in real-world data storage systems to significantly reduce costs.

In the following section, I will provide a brief introduction to the architecture of RocksDB and a review of memory systems. In section 3, I will go over the work that I have done and changes made to the RocksDB code. In section 4, I will explain how the evaluation has been done and using what tools. In section 5, I discuss the results achieved from the evaluations.

2 BACKGROUND

Before we get into changing the structure of RocksDB, it is worthwhile to acquaint ourselves with the architecture of the database and go over some memory systems concepts.

2.1 Memory Systems

Over the past few decades, computer systems have maintained a memory-storage hierarchy based on the principle of locality, keeping frequently accessed data closer to the CPU and getting them faster to the registers to work on [6].

In every modern memory storage system, there are several layers of memory. These layers of memory range from registers (placed on the CPU) and caches (feeding the data to the registers) to DRAMs and non-volatile storage (r.g. SSD, HDD, Tape, ...).

The memories closer to the CPU have the main responsibility of storing frequently accessed data and getting it to the CPU fast. The closer a memory is to the CPU the faster it should be. This demand for low latency comes at the cost of less storage and higher price ($/GB), such that as we get farther from the CPU the memories get slower but bigger and less expensive.

Throughout the years CPU speeds have continued to increase by adding more cores and threads and the caches have improved in parallel, providing frequently used data at a faster pace. On the other hand, the capacity, price, and speed of volatile memories (DRAM) and non-volatile storage systems (SSDs or HDDs) have not kept up and have become a bottleneck.

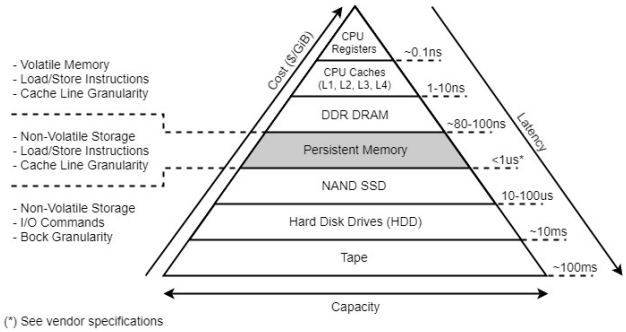

Figure 1: Memory Hierarchy: PMEM adds a new hybrid level to the memory hierarchy [intel]

Persistent Memory (PMEM) provides a new entry in the memory-storage hierarchy, as depicted in Figure 1. PMEMs are a much-improved hybrid of DRAM and external storage systems, as they combine the capacity of storage systems with a speed rate of main memory, not to mention they are non-volatile.

To begin with, PMEMs, with a maximum capacity of 512 GB for a single module, are much larger than DRAMs which have 64GB for a single module [9]. Furthermore, in terms of unit price, PMEM's cost per gigabyte is about half of the ordinary memory [8]. PMEMs also have a latency performance advantage of

1–2 orders of magnitude over ordinary SSDs [9]. Applications can access persistent memory as they do with traditional memory, eliminating the need to page blocks of data back and forth between memory and storage.

Even though PMEMs have many advantages, incorporating them into systems is not straightforward. For example, equating a real NVM with a slow DRAM can be complicated [3]. For one thing, NVM hardware has a lot of discrepancies compared to DRAM. Moreover, despite NVMs' high capacity compared to DRAM, they have up to 7× lower bandwidth and up to twice the latency [2], in addition, OS-based systems have overheads that prevent them from scaling to the capacity of NVMs. Thus novel main memory management systems have to be used in order to balance these trade-offs to provide high memory performance.

Thus incorporating PMEM in an application does not automatically result in improved performance. PMEMs can be beneficial only if there is a coordination between different hardware and software components and the resources are well managed [3].

2.2 RocksDB

RocksDB is an embeddable persistent KV-store for fast storage. There are a variety of components in RocksDB based on different data structures that help improve its performance, however, the three basic constructs of RocksDB are memtable, block cache, and logfile [10].

2.2.1 Memtables

Memtable is an in-memory data structure buffering data until flushed to the SST files; they can be considered an in-memory write buffer [10]. Memtables serve both read and write [10]. Hashtable-based linked lists or skiplists, inline skiplists, and vectors are some examples of the predefined data structures in RocksDB that can be used to back a memtable, however, the default implementation (and what we are using) is based on skiplists [10].

2.2.2 Block Cache

RockDB uses block caches to store data, as uncompressed blocks, in the main memory for reads. In the experiments, I am using LRU caches, where each shard of the cache maintains its own LRU list and hashtable for lookups. In RocksDB, all LRU caches have exclusive mutexes within to protect writes to the LRU lists, done in both lookup and insert operations. In addition to LRUcache, RocksDB has a ClockCache. Both types of cache types are sharded to mitigate lock contention.

2.2.3 Write Ahead Log(WAL)

The Write Ahead Log or WAL is used to keep track of the changes made in the DB. Every update to RocksDB is written WAL on disk as well as the memtable, and the WAL records all writes until it reaches its size limit [10].

2.2.4 RocksDB Write Path

In RocksDB a write request is first written to the active memtables and optionally stored in the logfile (WAL). Once a memtable is full, it becomes immutable (read-only memtable) and gets replaced by a new memtable. Then a background thread will flush the content of the memtable into an SST file on the storage system, after which the memtable and the corresponding log file can be destroyed [10]. The data in an SST file is lexicographically sorted to facilitate easy key lookups and sequential scans. In order to maintain a reasonable size, the SST file undergoes periodic compactions. To save on random reads, RocksDB keeps a bit array (Bloom filters) for every memtable and SST file [12]. Figure 2 demonstrates the write process in RocksDB.

2.2.5 Read Path in RockDB

When a read request comes in, the DB needs to pull the data into the block cache. If the data is not already available in the block cache, a read from other sources will occur starting with memtables [13]. Since the key-value pairs are first written to active memtables, they contain the newest data. Read requests initially check the active memtables for the key. If the key is not in the active memtable, the request checks the read-only memtable from the newest to the oldest

flushed. If the key is not found in any of the memtables, the SST files (using Bloom filters) are checked on the disk. If the key-value pair is found in the SST files then the compressed block containing the key will be retrieved from the SST files, the block will then be uncompressed and loaded in the block cache [10, 12]. Figure 3 demonstrates a simplified version of the read process in RocksDB.

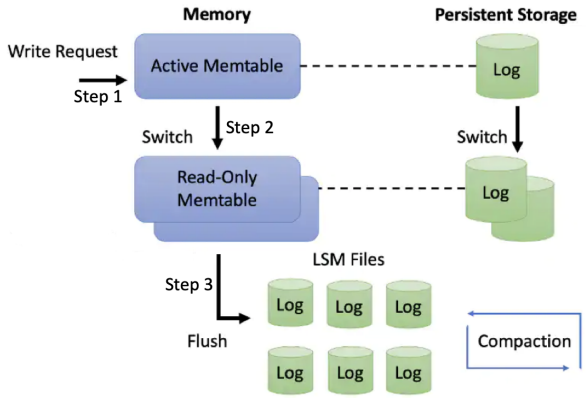

Figure 2: RocksDB Write Path

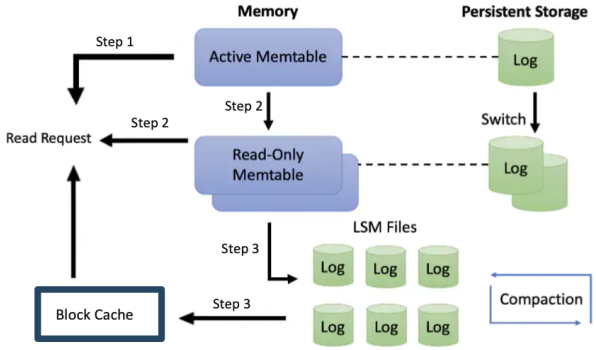

Figure 3: RocksDB Read Path

2.3 Memory Allocation

RocksDB has some pre-defined allocations classes that are used in the rest of the code. These allocation files are an implementation of the abstract interfaces *allocator.h* or *memory_allocator.h* and rely on some external memory allocators (e.g. jmalloc, memkind).

To allocate a RocksDB component it is essential to use an allocation class that implements one of the allocator interfaces. Depending on the functionality of each component they either use a *memory_allocator* or an *allocator*. For example, memtables use *Arenas* which are an instance of the *allocator* interface. *Arenas* allocate a block with a predefined block size for a request of small blocks but mallocs a block if the block size exceeds a certain predefined size. On the other hand, block cache uses an instance of *memory_allocator*.

Non-volatile dual in-line memory modules (NVDIMMs), for example, Optane DC PMM, are exposed by the operating system as devices on which users can create file systems. Therefore, we need a way to use the memory exposed through files in applications. This means that it is not possible to directly store a memtable or block cache on the PMEM, and they have to be stored in file systems that are mounted on the PMEM. There are a few libraries that help with allocation on the PMEM, in my experiments I use *memkind* and *pmem*. The *pmem* library is built on *memkind*.

2.3.1 Memkind

Memkind is a memory allocator built on top of jmalloc that can be used to create volatile memory on a persistent memory region, preferably an Intel Optane DC PMM device.

Memkind is essentially a wrapper for jmalloc, such that jmalloc is responsible for the heap management and memkind just redirects jmalloc's memory requests to a different place. Memkind uses memory-mapped files to create the perception of a volatile region.

To use memkind on an Intel Optane DC PMM, the persistent memory must first be provisioned into namespaces to create the logical device /dev/pmem. Moreover, a DAX-enabled filesystem should be mounted on the device [14].

When using memkind a temporary file is created on said DAX-enabled file system and memory-mapped into the application's virtual address space. The address space is used by jmalloc later on to allocate objects [11]. The file is automatically deleted when the program terminates, giving the perception of volatility. Figure 4 depicts how memkind uses file systems to allocate space on PMEM. It is also possible to allocate memory on DRAM and High-Bandwidth Memory with memkind.

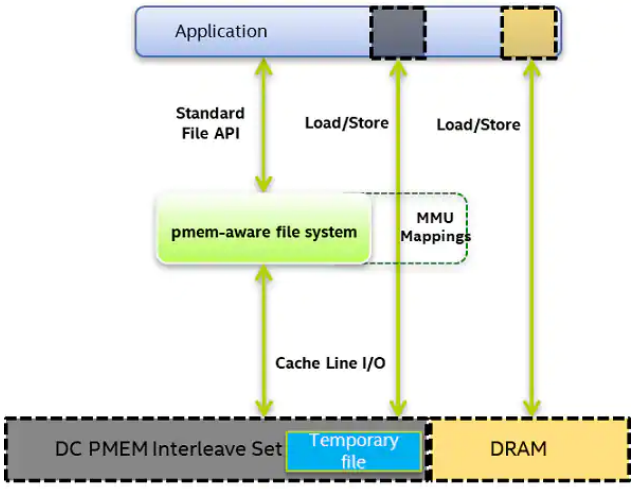

Figure 4: PMEM Allocation using memkind [Intel]

3 Contributions

In this research, I am primarily interested in exploring ways to have RocksDB's common data structures in PMEM without affecting the performance of the application. I am specifically interested to see if there is a way to allocate a section of the block cache and memtable used by RocksDB in the PMEM.

To run measurements and test the DB, I have tried three different combinations for the placement of block cache and memtable:

1. Block Cache and Memtables both in DRAM: this is the default setting of RocksDB
2. Block Cache in PMEM but Memtables in DRAM: In this case, we will use the default implementation for the memtables but have to change some code to allocate the Block Cache in the PMEM
3. Block Cache and Memtables both in PMEM: The implementation of both data structures should be changed so they get allocated in PMEM

I was initially planning on experimenting with having only a portion of the memtable and block cache in the PMEM, for example, 50% of the block cache and 40% of the memtable. However, due to the lack of time and the complexity of the code, that was not possible.

3.1 Implementation

A few things need to take place to successfully move a RocksDB data structure from DRAM to PMEM. For one thing, the allocation method for the data structure needs to be changed such that it is placed in the PMEM, this is done using the *memkind* library. Furthermore, the supporting functions that work on the memory, for example, insert, remove, deallocate, etc., need to be changed accordingly.

As mentioned in the previous section, to allocate memory on PMEM, we need to create a temporary file on a DAX-enabled file system. *Memkind* provides function and structs that helps with that. Listing 1 presents memkind functions that can be used to allocate a partition on the PMEM. In the sniped *pmem_kind* is a pointer to a PMEM partition. *memkind_create_pmem()* creates a temporary file at the path specified by pmem_dir (must be a DAX enabled directory) with the size of *max_size* bytes and set a pointer to the partition in *pmem_kind* [4, 14, 16].

```
struct memkind *pmem_kind = NULL;

int memkind_create_pmem(const char *pmem_dir,
size_t max_size, memkind_t *pmem_kind);
```

Listing 1: memkind function used to create a memory partition on PMEM

Now that we have a pointer to a PMEM partition, we need to allocate our data structures on it. For this, we use the function defined by *memkind* shown in Listing 2. This function allocates *size* bytes of uninitialized *kind* memory [4, 14, 16].

```
void *memkind_malloc(memkind_t kind, size_t size)
```

Listing 2: memkind function used to allocate space on a memory partition

In the RockDB implementation, to allocate the block cache in PMEM, I have modified the *memkind_kmem_allocator.cc* file.

```
struct memkind *pmem_kind = NULL;
int err = memkind_create_pmem("/tmp/", PMEM_MAX_SIZE, &pmem_kind);

namespace ROCKSDB_NAMESPACE {

void* MemkindKmemAllocator::Allocate(size_t size) {
  void* p = memkind_malloc(pmem_kind, size);
  if (p == NULL) {
    throw std::bad_alloc();
  }
  return p;
}

...
```

Listing 3: A sniped of *memkind_kmem_allocator.cc* code used to allocate block cache in PMEM

Listing 3 shows a sniped of code from the modified *memkind_kmem_allocator.cc* file. In RockDB block caches can be allocated using the *memkind_kmem_allocator* which implements *memeory_allocator.h*. In the code, memkind functions is being used to allocate a global PMEM partition which would imitate a block cache on PMEM. After that, the allocate function is used to allocate a block of *size* bytes on the block cache and a pointer to the block is returned to be used in the other functions. In the rest of the *memkind_kmem_allocator.cc* code, other *memkind* defined functions are being used to manage the PMEM partition for example to deallocate an allocated block.

Changing the allocation for the memtables was more challenging. Given that the memtables are based on a specific data structure (e.g. skiplist, hashtable, …) and deal with constant read and writes, changing their allocation is much more complicated.

In my experiments, I used the skiplist based memtables, which means that the memtable is essentially a warper for a skiplist. In RocksDB, skiplists are based on *Arenas*. *Arenas* are a class of memory allocators that essentially allocate a *Vector* as their main storage component but provide multiple functions to manage and work on the *Vector*.

In my implementation, I changed the *arena.h* and *arena.cc* code such that they have the same functionality as before but only the main storage component (a vector) would be allocated in PMEM and other operations would have to access PMEM. I used the *pmem_allocator* library to allocate a vector in PMEM and changed the supporting functions accordingly. Listing 4 shows the way the Vector used by the Arena is getting allocated using the pmem_allocator [17, 18]. *libmemkind::pmem::allocator<T>* is used with STL containers to allocate persistent memory and is based on the memkind library. I use the *libmemkind::pmem::allocator* to create a PMEM allocator *alcvec* on the *pmem_directory*. I then use *alcvec* to allocate a vector *blocks_* which will be used by the memtable to store data.

```
size_t pmem_max_size = 1024 * 1024 * 32;
const char *pmem_directory = "/tmp/";

libmemkind::pmem::allocator<char*>
alcvec{pmem_directory, pmem_max_size};

std::vector<char*,
libmemkind::pmem::allocator<char*>>
blocks_{alcvec};
```

Listing 4: allocating a vector in PMEM used by memtable, in the *arena.h* file

The complete code change and implementation can be found on the project's Git repository.

3.2 Splitting Between DRAM & PMEM

My initial intention was to also experiment with splitting the block cache and memtable between DRAM and PMEM. The only way this could be possible is to simultaneously manage two data structures for each component, one on each memory. For example, two vectors for the memtable, one on DRAM and one on PMEM. The addition of a new data structure would require additional logic and code to be added to RocksDB to manage the database, which was too complicated to be done on time. For example, the key insertion policy would have to change to account for a second memtable and block cache on the PMEM.

3.3 Challenges

Coding the PMEM allocation was quite challenging for a few reasons:

1. PMEM is new: as mentioned before PMEM is quite new and there are not a lot of libraries available to work with it. Figuring out how exactly to use these new libraries (*memkind* and *pmem*) and how to incorporate them in the RocksDB code was a bit time-consuming.
2. Working with memory is hard: working with memory is not trivial as I am sure everyone who has run into segmentation faults and core dumps knows.

As an example, I initially had some trouble linking the memkind library to the RockDB code, after some investigation, it turned out that there was a bug in the RockDB makefile code that did not recognize the memkind library. Furthermore, I received memory allocation errors when trying to use the predefined memkind allocator that used the MEMKIND_DAX_KMEM allocation type, allocate from the closest persistent memory NUMA node, which was fixed by making some changes to the RocksDB source code and using a different memkind function to allocate space on PMEM.

4 EVALUATION

In this section, we set out to answer the following:

1. What effects will the placement of memtable and block cache in PMEM have on the system's latency and throughput?
2. How will read and write latencies be affected by moving the block cache and memtable to PMEM?
3. Will the hit and miss ratios for block cache and memtable be affected as they are placed in PMEM?
4. Is it possible to place LSM components in PMEM, while maintaining the performance of having them in DRAM?

4.1 Evaluation Environment

The evaluations have been completed on a two-socket machine using Intel Xeon Gold 6240 processors containing 18 cores and 36 threads per socket. The machine contains a total of 3TBs of Intel Optane DC Persistent Memory and a total of 768 GBs of DRAM. I use RockDB version 6.26.1, gcc version 9.3.0, and memkind version 1.11.0.

4.2 Benchmarking

I use the benchmarking tool (db_bench) available in RocksDB, which is the main tool used to benchmark RocksDB's performance. db_bench supports many options and allows many different benchmarks and workloads to be tested.

4.3 Workload Configuration

For our benchmarking, I use a default set of parameters that is common between all the measurements done. These parameters include:

- **benchmarks = readrandomwriterandom:** we run the read-random write-random benchmark consisting of N threads (which we will define) doing random-read, random-write
- **readwritepercent=50:** we have a 50/50 % ratio for reads and writes
- **value_size=1024**: size of each value is set to be 1024B

I have added some options to highlight the exact difference between the scenarios I am trying to

examine and isolate the block cache and memtable to their basic model:

- **enable_pipelined_write=false:** does not allow memtable writes to be pipelined, isolating the memtables write performance
- **allow_concurrent_memtable_write=false:** does not allow multi-writers to update mem tables in parallel
- **Memtable_use_huge_page=true:** allows using huge pages in memtable

There are 24 workloads in total in 3 scenarios. The scenarios are

I. Default RocksDB architecture: memtable and block cache both in DRAM

II. Block cache in PMEM and memtable in DRAM

III. Block cache and memtable both in PMEM

Each scenario is run with 8 different parameter settings which is a permutation of 8 and 16-byte key_size and threads going from 1 to 8 in powers of two. Moreover --use_cache_memkind_kmem_allocator is used to run experiments with the block cache in PMEM.

5 DISCUSSION

The benchmarks set for the configurations follow the same pattern, so to keep the evaluation concise, I will use the benchmark results from a variety of 4 different configurations. Figure 5 shows the throughput (ops/sec) for each configuration and Figure 6 shows the latency (micros/op). As seen in the figures, each group of bars shows the data for a key-thread configuration where K is the key size and T is the number of threads used, for example, K16_T1 is the case with 16-byte keys and 1 thread. Moreover, the blue bars show the case where neither memtable nor block caches are in PMEM (both in DRAM), red shows when only the block cache is in PMEM, and green when both are in PMEM.

From figure 6, we can observe, for all four configurations, that as we put both the memtable and block cache in PMEM the latency almost doubles. This sudden increase in latency is not surprising given that for any read or write of any type of block the PMEM has to be accessed which will undoubtedly affect the latency. However, the latency between the default configuration (both in DRAM) and the configuration with only the block cache in PMEM is quite similar and in many cases indistinguishable. This is great given that it means by putting the entire block cache in PMEM we are not losing any performance but gaining a part of the valuable capacity of the DRAM.

Figure 5, showing the throughput, strengthen the findings from the latency graphs. The throughput early halves as we put both the memtable and block cache in PMEM but for the other two configurations, the throughput is almost the same, with the config of block cache in PMEM having about a 1% lower throughput on average. It's clear that latency and throughput have a direct correlation with the key size and number of threads.

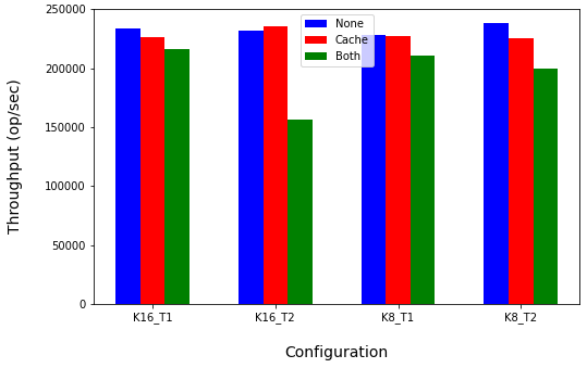

Figure 5: Throughput Benchmarks

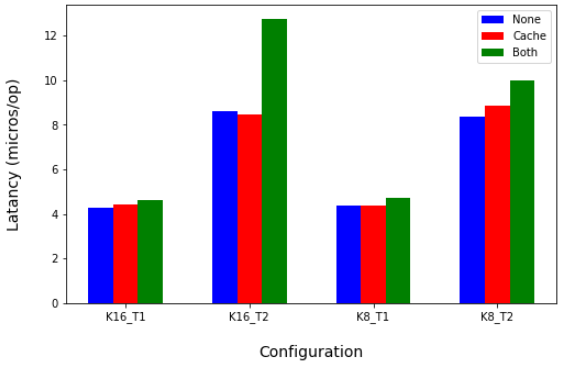

Figure 6: Latency Benchmarks

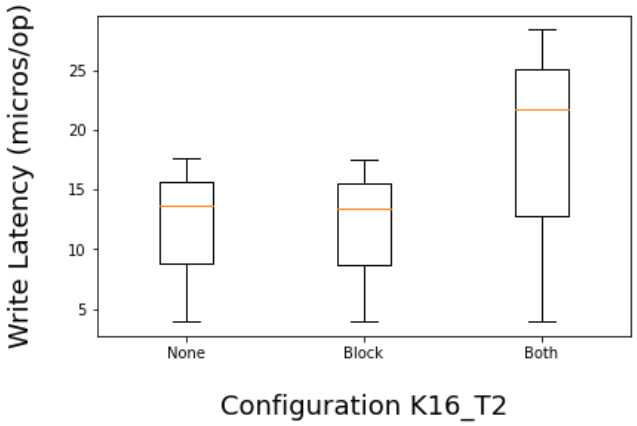

Figure 7: Write Latency Percentile (K16_T2)

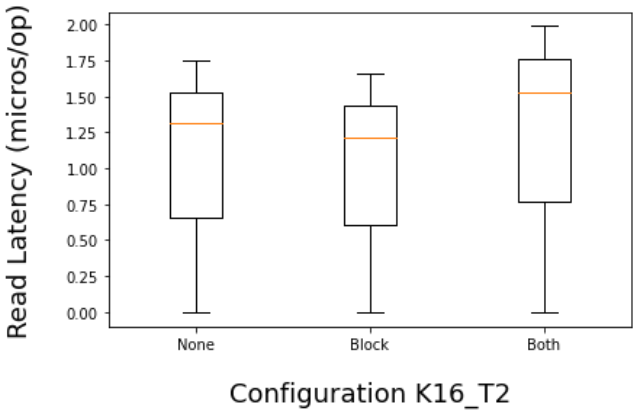

Figure 8: Write Latency Percentile (K16_T2)

Figures 7 and 8, show the write and read latency percentiles of the K8_T2 configuration. As seen in Figure 7, as the block cache is moved to the PMEM the write latency is not changed, this is because writes are done in the memtable and as the memtable is still in DRAM a fast write latency is expected. But, as the memtables is moved to the PMEM the write latency increase.

Given that the reads are done from either the block cache or the memtables, we would expect an increase in read latency as both the block cache and memtable are moved to the PMEM. In Figure 8, there is a slight increase in read latency as both the block cache and memtable are moved to PMEM, however, the increase is less than expected. The small increase in read latency can not be entirely attributed to a specific key size and number of threads, given that the other benchmarks follow the same read latency pattern. The reason why the read latency is not increasing at a higher rate is not entirely clear to me at the moment, however, given that the implementation and PMEM allocation used in the code follow the memkind documentations I'm not entirely sure it can be related to the way the memory is allocated. I believe the reason behind the small increase in read latency is due to the logic used by RocksDB to optimize read performance.

Figure 7 demonstrates the number block cache and memtable hits and misses if we were to use a higher number of threads. As seen in Figure 7, the rate for memtable increases as the data structures are placed in the PMEM. Given that the size of the memtable and block cache are set and do not change the change in a hit and miss number can not be attributed to the memory placement of the data structures. But since we are using read random write random to benchmark the database, the difference in the hit and miss numbers might be due to the random patterns of filling the database and the seed of the random number generator used. For K8_T2 the data amount is so low that the entire effort has gone into filling the memtables and not much has to be retried after filling the memtables, which explains the high miss count and low hit count.

Based on the results, the ideal scenario is to keep the block cache (or at least a larger proportion of it) in PMEM but still, keep the memtable in DRAM to enable fast access.

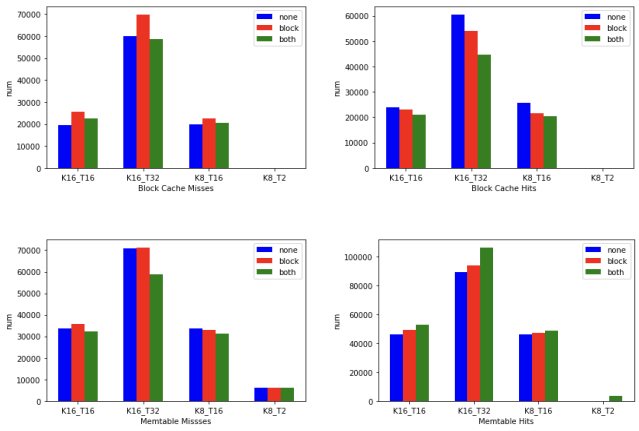

Figure 7: Miss and Hit Numbers for the Block Cache and Memtable

6 FUTURE DIRECTIONS

Unfortunately, due to the limited time I was not able to experiment with having sections of the block cache and memtable in the PMEM. In the future, I would like to explore if there is a way to split the block cache and memtable between the PMEM and DRAM in a way to optimize the usage of the main memory and performance of RocksDB simultaneously. Furthermore, I would be interested to experiment with the effects of different data structures and access patterns used for the memtable and block cache. For example, keep 60% of the LRU block cache in PMEM which holds compressed blocks, etc.

7 CONCLUSION

In this research, I analyze storing components of LSM-tree based key-value stores specifically memtable and block cache in persistent memory (PMEM), rather than DRAM. To this end, I ran a set of benchmarks on RocksDB, with multiple workloads and memory placements for memtable and block cache. The evaluation shows that is possible to place the block cache on the PMEM with minimal influence on the application's performance. The results show that placing only the block cache on the PMEM has little to no effect on the application's latency and throughput. Placing the block cache in PMEM, with the memtable in DRAM, increases the latency by about 15%. Furthermore, placing only the block cache on PMEM increases the mean write latency by less than 1% on average. Placing the block cache on PMEM comes with the advantage of freeing up space in the range of tens of MBs on the DRAM. On the other hand, placing both the block cache and memtable will dramatically increase the latency (by 2x) and decrease the performance of the application.

ACKNOWLEDGMENTS

I would like to express my appreciation to Dr. Oana-Balmau, my research supervisor, for their patient guidance, encouragement, and constructive critiques of this research work

REFERENCES

[1] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquín, and Donald Kossmann. 2017. Fast scans on key-value stores. *Proc. VLDB Endow.* 10, 11 (August 2017), 1526–1537. DOI:https://doi.org/10.14778/3137628.3137659

[2] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21). Association for Computing Machinery, New York, NY, USA, 392–407. DOI:https://doi.org/10.1145/3477132.3483550

[3] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21). Association for Computing Machinery, New York, NY, USA, 424–439. DOI:https://doi.org/10.1145/3477132.3483589

[4] Lyon K, Upadhyayula U. Use Memkind to Manage Large Volatile Memory Capacity on Intel®... Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/use-memkind-to-manage-volatile-memory-on-intel-optane-persistent-memory.html. Published 2021. Accessed November 29, 2021.

[5] Biesek M. pmem.io: Memkind support for KMEM DAX option. Pmem.io. https://pmem.io/2020/01/20/memkind-dax-kmem.html. Published 2021. Accessed November 29, 2021.

[6] Maximilian Böther, Otto Kißig, Lawrence Benson, and Tilmann Rabl. 2021. Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs. In Proceedings of the 17th International Workshop on Data Management on New Hardware (DAMON'21). Association for Computing Machinery, New York, NY, USA, Article 7, 1–8. DOI:https://doi.org/10.1145/3465998.3466010

[7] Introduction - Persistent Memory Documentation. Docs.pmem.io. https://docs.pmem.io/persistent-memory/getting-started-guide/introduction. Published 2021. Accessed November 29, 2021.

[8] Shilov A. Pricing of Intel's Optane DC Persistent Memory Modules Leaks: From $6.57 Per GB. Anandtech.com. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks. Published 2021. Accessed November 29, 2021.

[9] Intel® Optane™ Persistent Memory. Intel. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Published 2021. Accessed November 29, 2021.

[10] facebook/rocksdb Wiki. GitHub. https://github.com/facebook/rocksdb/wiki. Published 2021. Accessed November 29, 2021.

[11] Memkind. Memkind.github.io. http://memkind.github.io/memkind/. Published 2021. Accessed November 29, 2021.

[12] BORTHAKUR D, CADONNA B. How to Tune RocksDB for Your Kafka Streams Application. confluent.io/. https://www.confluent.io/ja-jp/blog/how-to-tune-rocksdb-kafka-streams-state-stores-performance/. Published 2021. Accessed November 29, 2021.

[13] Sylvester P. Exposing MyRocks internals via system variables: Part 5, Data Reads. Official Pythian® Blog. https://blog.pythian.com/exposing-myrocks-internals-via-system-variables-part-5-data-reads/. Published 2021. Accessed November 29, 2021.

[14] pmem.io: Introduction to libmemkind. Pmem.io. https://pmem.io/2020/01/20/libmemkind.html. Published 2021. Accessed November 29, 2021.

[15] Use Memkind to Manage Large Volatile Memory Capacity on Intel®... Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/use-memkind-to-manage-volatile-memory-on-intel-optane-persistent-memory.html. Published 2021. Accessed November 29, 2021.

[16] MEMKIND. Memkind.github.io. http://memkind.github.io/memkind/man_pages/memkind.html. Published 2021. Accessed November 29, 2021.

[17] PMEMALLOCATOR. Memkind.github.io. http://memkind.github.io/memkind/man_pages/pmemallocator.html. Published 2021. Accessed November 30, 2021.

[18] memkind/pmem_allocator_tests.cpp at master · memkind/memkind. GitHub. https://github.com/memkind/memkind/blob/master/test/pmem_allocator_tests.cpp. Published 2021. Accessed November 30, 2021.