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

Author: Changjun Zhou, Supervisor: Prof. Oana Balmau

LSM-tree based key-value store's high throughput and space efficiency made them an essential component of modern databases. However, key-value stores usually take up to a significant amount of expensive memory to operate at the desired speed. Given the cost of DRAM and the demand for it by many LSM-tree key-value store applications, performance improvement of LSM-tree key-value is not very cost-effective. Therefore, we should seek a DRAM alternative to lower the cost while introducing as little performance decrease as possible. Novel byte-addressable permanent memory (PMEM) offers data persistence at speed close to DRAM. In this paper, we used RocksDB - a database that uses LSM-tree and studied the placement of Block Caches and Memtable's placement on DRAM and PMEM. We discovered that placing the entire Block Cache on PMEM / placing the entire Block Cache and memtable on PMEM decreases the RocksDB benchmark's throughput but alleviates MBs to GBs DRAM usage. Note that this paper is written as a smooth introduction to our research project and to bridge the knowledge gap for new researchers.

Introduction

Persistent Memory

Persistent memory(PMEM) is also called storage class memory. PMEM could persists the data across power loss. PMEM are byte-addressable, which means programs could use PMEM in place. Persistent memory is fast enough to access directly from the processor without stopping to do the Block I/P required for traditional storage. PMEM, with a maximum capacity of 512 GB for a single module, are much larger than DRAM which have a maximum of 64GB for a single module. Furthermore, PMEM's cost per gigabyte is about half of that of DRAM in terms of uni price. In terms of latency, PMEM is 1-2 orders of magnitude faster than SSDs. However, PMEM has up to 7x lower bandwidth and up to twice the latency compared to DRAM [4].

LSM tree basic structures

LSM-tree is a widely used data structure for key-value pair databases. LSM-tree greatly improved write throughput and latency by properly using the characteristics of DRAM and disks. There are two major parts that supports the functioning of LSM-tree. The first part are the in-memory components - Block Cache and Memtables. More details about these two data structures will be introduced later. The purpose of these in-memory components is to enable fast writing for LSM-tree. All income writes are directly added to Memtables as logs. The Block cache is a in-memory cache of the database, so that users could read frequently accessed data directly from memory.[1] LSM-tree periodically transfer data in Memtable into disk. Data is stored as files on disk, these files are called SSTFiles. The in-disk component of LSM-tree is spitted into different tiers, the deeper the tiers are, usually the older the data is and the larger the tiers are. When a tier is full, a merging operation is performed to move data into the next tier. Note that the merging operation is very resource consuming. There are different kind of merging policy to make the merging process impact the performance of LSM-tree less. For more details about LSM-tree, see this article [2].

Block Cache

RocksDB use Block Cache to store uncompressed blocks in memory to accelerate reads. RocksDB provide two kinds of Block Cache, LRUCache and ClockCache. Out of the box, RocksDB use LRUCache with an 8MB capacity [3]. RocksDB will try to search in the Block Cache first when a read request comes in. Then, if the data is not already available in Block Cache, a read from other sources will occur, starting with Memtable.

Memtable

MemTable is an in-memory data structure contains newly-written data temporarily, before those data are flushed to disks. It involves in both reading and writing - new writes always insert data to Memtable, and reads have to query Memtable before reading from SST files. Once a Memtable is full, flushing mechanism is triggered and the content of the memtable will be moved to disk to persist[3]. Note that in this research, we are investigating the behaviour of placing Block cache and Memtable into PMEM. Also, note that Memtable and Block Cache are both in-memory data structures.

Memory Allocation

RocksDB has some pre-defined allocations classes used in the rest of the code. These allocation files implement 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, Memtable uses Arenas, 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 specific 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 store a Memtable or Block Cache on the PMEM directly, and they

have to be stored in file systems that are mounted on the PMEM. There are a few libraries that help with the allocation of the PMEM. In my experiments, I use memkind and PMEM. The PMEM library is built on memkind.

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. When using memkind, a temporary file is created on a 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. The file is automatically deleted when the program terminates, giving the perception of volatility. It is also possible to allocate memory on DRAM with memkind.

Methods

Note that this research aims to investigate performance changes when combining RocksDB with PMEMs. To achieve this, we set up different scenarios for RocksDB's Block Cache and Memtable. Scenario one: Both Block Cache and Memtable are in memory. This is the default setting of RocksDB. Scenario two: Block Cache stays in PMEM while Memtable stays in PMEMs. Therefore, we changed the Block Cache codes to make it use PMEM. Scenario three: Both Block Cache and Memtable are in PMEM. Therefore, we changed the Block Cache and Memtable code to make them use PMEM.

Implementation

First, we need to mount the PMEM as a DAX-enabled file system. Then we need to change the code of Block Cache and Memtable to make them use PMEM instead of DRAM, specifically the code that has to insert, remove, deallocate etc., functionalities. This could be done by using memkind. The details of the implementations can be found at https://gitlab.cs.mcgill.ca/discs-lab/pmem-caching-rocksdb. It is carefully commented, and the readme file contains everything readers need to know to set up the testing environment.

Evaluation

In this section, we want to answer this question: What effects will the placement of Memtable and Block Cache in PMEM have on the system's throughput?

Environment

The evaluation has been completed on a two-socket machine using Intel Xeon Gold 6240 processors containing 18 cores and 36 threads per socket. The machine contains 3TBs of Intel Optane DC persistent memory and a total of 768GBs of DRAM. I use RocksDB version 6.26.1, GCC version 9.3.0, and memkind version 1.11.0.

Benchmarking

I use the benchmarking tool db_bench available in RocksDB, the primary tool used to benchmark RocksDB performance. For our benchmarking, I use the parameters below:

benchmarks=readrandomwriterandom
value_size=1024
disable_wal=true
cache_size=512000000
compression_ratio=1
readwritepercent=50
use_existing_db=0
histogram=1
statistics=1
read_random_exp_range=0.0
max_write_buffer_number=1
memtable_use_huge_page=true
enable_pipelined_write=false
allow_concurrent_memtable_write=false
write_buffer_size=10485760
num=1000000000
duration=900
stats_interval_seconds=1
key_size=16 threads=16
use_cache_memkind_kmem_allocator

We added some options to highlight the exact difference between the scenarios we are trying to examine and isolate the Block Cache and Memtable to their basic model:
enable_pipelined_write=false
allow_concurrent_Memtable_write=false
Memtable_use_huge_page=true

There are nine workloads in three scenarios. The scenarios are:
1 Memtable and Block Cache both in DRAM (DRAM scenario)
2 Block cache in PMEM and Memtable in DRAM (PMEM scenario)
3 Block cache and Memtable are both in PMEM (PMEM scenario)

Each scenario is run with 3 different parameter settings which 1% read/ 99% write, 50% read and 50% write, lastly, 90% read and 10% write. The 90% read and 10% write is the default workload of RocksDB's Benchmark tool. The 1% read/ 99% write aims to simulate a pure write workload.

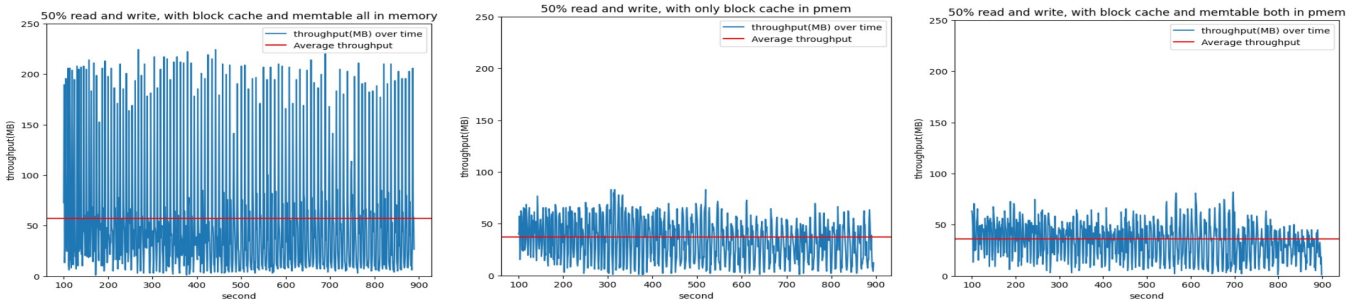

Fig. 1. RocksDB throughput for 50% read and write for Scenario 1 **Fig.1.1** RocksDB throughput for 50% read and write for Scenario 2 **Fig.1.2** RocksDB throughput for 50% read and write for Scenario 3

Discussions

This section will discuss the benchmarking result of the nine workloads in 3 scenarios. First, note that since the bandwidth and latency of PMEM are smaller than those of DRAM, it is evident that the benchmark result for using PMEM would be slower than using only DRAM. However, we are interested in how much slower it is.

50% read and write

To make the comparisons concise, I will first compare the 50% read/ 50% write case throughput for three scenarios. Figure 1 is the benchmark result of 50% read and writes, with Block Cache and Memtable all in DRAM. Figure 1.2 is the benchmark result of 50% read and write, with Block Cache in PMEM and Memtable in DRAM. Figure 1.3 is the benchmark result of 50% read and write, with both Block Cache and Memtable in PMEM.

The result is very straightforward. The average throughput of 50% read and write, with Block Cache in PMEM and Memtable in DRAM is the same as that of 50% read and write, with both Block cache and Memtable in PMEM — 36 MB/s. While the average throughput of 50% read and write, with Block Cache and Memtable in memory, is around 57 MB/s, which is 36% more than the other two scenarios' throughput where PMEM was involved, considering all in-memory components are using DRAM.

I also monitored DRAM and PMEM bandwidth when running the benchmarks above. We use this as a sanity check to make sure that the program is actually using PMEM and running correctly by comparing the bandwidth of DRAM and PMEM with the expected bandwidth.

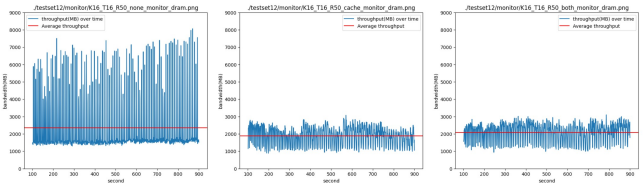

Fig. 2. From left to right: Bandwidth of DRAM during benchmarking for Fig1, Fig1.1 and Fig1.2

Figure 2 shows the bandwidth of DRAM in the computer during the benchmark for 50% of reading and 50% of writing cases. What these graphs show makes logical sense. When we have both Block Cache and Memtables in DRAM, the average bandwidth of DRAM is two times more than the other two cases's bandwidth where PMEM was involved.

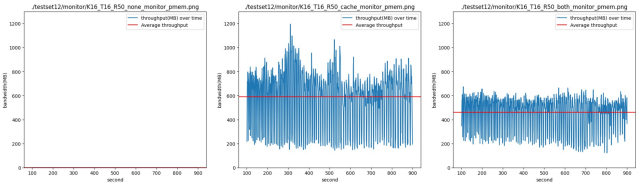

Fig. 3. From left to right: Bandwidth of PMEM during benchmarking for Fig1, Fig1.1 and Fig1.2

Figure 3 shows the bandwidth of PMEM for 50% of reading and write cases. Note that the PMEM usage is zero for the first graph of Figure 3 because both the block cache and Memtable are in DRAM, so PMEM was not used. The average bandwidth of PMEM for the other two PMEM scenarios are close, and it remains close after I do the benchmarks several times. This shows that when memtable is in PMEM, memtable does not use a noticeable amount of PMEM bandwidth.

1% read and 99% write

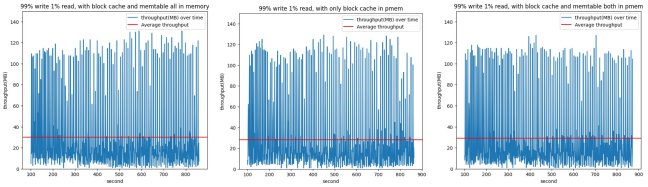

Fig. 4. Scenario 1, RocksDB throughput for 1% read and 99% write for Scenario 1 **Fig.4.1** Scenario 2 **Fig.4.2** Scenario 3

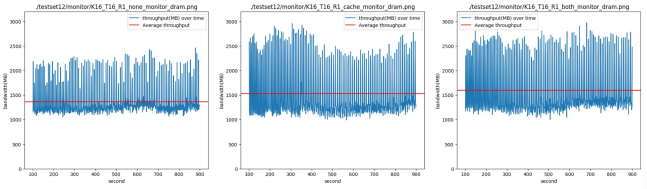

Fig. 5. From left to right: Bandwidth of PMEM during benchmarking for Fig4, Fig4.1 and Fig4.2

Figure 4, 5, and 6 shows that in the case of 1% read and 99% write (write dominant), the benchmark's throughput of three scenarios are close, with the scenario "Block Cache

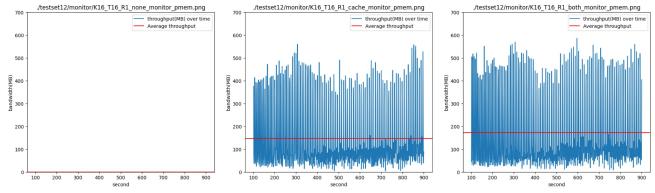

Fig. 6. From left to right: Bandwidth of PMEM during benchmarking for Fig4, Fig4.1 and Fig4.2

and Memtable all in DRAM" having an average throughput of 30MB/s, "Block Cache in PMEM and Memtable in DRAM" have an average throughput of 28MB/s, and "Block Cache and Memtable in PMEM" have an average throughput of 29MB/s. The two PMEM scenarios have almost the same throughput, and they are only around 7% slower than the DRAM scenario.

90% read and 10% write

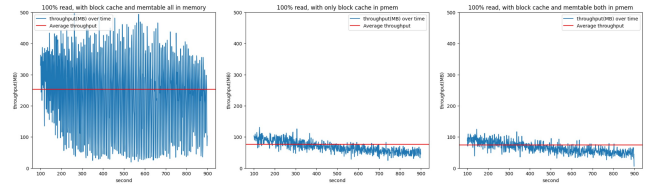

Fig. 7. Scenario 1, RocksDB throughput for 90% read and 10% write for Scenario 1 **Fig.7.1** Scenario 2 **Fig.7.2** Scenario 3

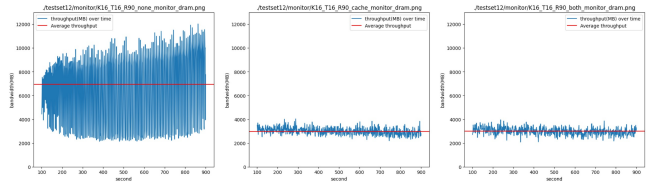

Fig. 8. From left to right: Bandwidth of PMEM during benchmarking for Fig7, Fig7.1 and Fig7.2

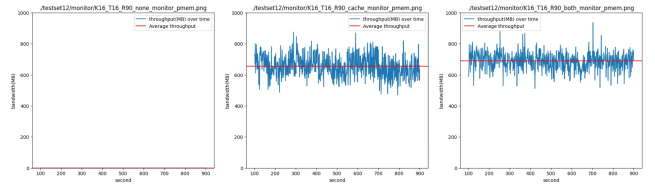

Fig. 9. From left to right: Bandwidth of PMEM during benchmarking for Fig7, Fig7.1 and Fig7.2

Figure 7,8 and 9 show that in the case of 90% read and 10% write, the throughput of "Block Cache and Memtable in DRAM" is 70% larger than the other two scenarios that use PMEM, with the scenario "Block Cache and Memtable all in DRAM" have an average throughput of 253 MB/s, "Block Cache in PMEM and Memtable in DRAM" have an average throughput of 77 MB/s, and "Block Cache and Memtable in PMEM" have an average throughput of 74 MB/s. So again, the last two PMEM scenarios that use PMEM have almost the same throughput, but the last two PMEM scenarios are a lot slower than the DRAM scenario.

I had some interesting observations when comparing the above nine workloads we had across three different scenarios.

First, using PMEM for RocksDB's in-memory component decreases its throughput. The throughput decrease ranges from 7% to 70%, depending on the type of workload. Typically, the throughput difference using PMEM compared to not using PMEM for in-memory components becomes smaller when the write/read ratio becomes larger. For example, for non-read heavy workloads where the read/write ratio is 0.5, the throughput of the two PMEM scenario is around 36% smaller than the DRAM scenario. The reason for this behavior might be that, when the workload is more write heavy, RocksDB spends more time on compaction and flushing, the decrease of read/write latency for Memtables and Block Cache become more negligible. When the workload is more read-heavy, especially in our particular workload where the block cache size is set to 512 MB, the hit rate of block cache is high enough that the cache hit rate is more than 86%. Most data were directly retrieved from Block Cache in PMEM or DRAM. This 86% might come from the latency and bandwidth difference between PMEM and DRAM.

Conclusions

In this research, I analyzed the behaviour of RocksDB after placing Block Cache or Block Cache and Memtable into PMEM. The result shows that using PMEM for RocksDB's in-memory component decreases its performance from 7% to 70%. The performance decrease is more apparent when the workload becomes more read-heavy. However, the result also shows that for all kinds of workloads, the throughput of RocksDB stays the same for Block Cache in PMEM and Block cache + Memtable in PMEM.

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.

Reference

[1] O'Neil, P., et al.: The log-structured merge-tree (LSM-tree). Acta Inf. 33(4), 351–385 (1996)

[2] Chen Luo, Michael J. Carey: LSM-based Storage Techniques: A Survey. arXiv:1812.07527 (2018)

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

[4] 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