Reducing DRAM Footprint with NVM in Facebook

Assaf Eisenman[1,2], Darryl Gardner[2], Islam AbdelRahman[2], Jens Axboe[2], Siying Dong[2], Kim Hazelwood[2], Chris Petersen[2], Asaf Cidon[1], Sachin Katti[1]

[1]Stanford University, [2]Facebook, Inc.

ABSTRACT

Popular SSD-based key-value stores consume a large amount of DRAM in order to provide high-performance database operations. However, DRAM can be expensive for data center providers, especially given recent global supply shortages that have resulted in increasing DRAM costs. In this work, we design a key-value store, MyNVM, which leverages an NVM block device to reduce DRAM usage, and to reduce the total cost of ownership, while providing comparable latency and queries-per-second (QPS) as MyRocks on a server with a much larger amount of DRAM. Replacing DRAM with NVM introduces several challenges. In particular, NVM has limited read bandwidth, and it wears out quickly under a high write bandwidth.

We design novel solutions to these challenges, including using small block sizes with a partitioned index, aligning blocks post-compression to reduce read bandwidth, utilizing dictionary compression, implementing an admission control policy for which objects get cached in NVM to control its durability, as well as replacing interrupts with a hybrid polling mechanism. We implemented MyNVM and measured its performance in Facebook's production environment. Our implementation reduces the size of the DRAM cache from 96 GB to 16 GB, and incurs a negligible impact on latency and queries-per-second compared to MyRocks. Finally, to the best of our knowledge, this is the first study on the usage of NVM devices in a commercial data center environment.

1 INTRODUCTION

Modern key-value stores like RocksDB and LevelDB are highly dependent on DRAM, even when using a persistent storage medium (e.g. flash) for storing data. Since DRAM is still about 1000× faster than flash, these systems typically leverage DRAM for caching hot indices and objects.

MySQL databases are often implemented on top of these key-value stores. For example, MyRocks, which is built on top of RocksDB, is the primary MySQL database in Facebook and is used extensively in data center environments. MyRocks stores petabytes of data and is used to serve real-time user activities across large-scale web applications [23]. In order to achieve high throughput and low latency, MyRocks consumes a large amount of DRAM. In our case, we leverage a MyRocks server that consists of 128 GB DRAM and 3 TB of flash. Many other key-value stores are also highly dependent on DRAM, including Tao [9] and Memcached [24].

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04.
https://doi.org/10.1145/3190508.3190524

	DRAM	NVM	TLC Flash
Max Read Bandwidth	75 GB/s	2.2 GB/s	1.3 GB/s
Max Write Bandwidth	75 GB/s	2.1 GB/s	0.8 GB/s
Min Read Latency	75 ns	10 μs	100 μs
Min Write Latency	75 ns	10 μs	14 μs
Endurance	Very High	30 DWPD	5 DWPD
Capacity	128 GB	375 GB	1.6 TB

Table 1: An example of the key characteristics of DDR4 DRAM with 2400 MT/s, NVM block device, and 3D TLC flash device in a typical data center server. DWPD means Device Writes Per Day.

As DRAM faces major scaling challenges [19, 22], its bit supply growth rate has experienced a historical low [17]. Together with the growing demand for DRAM, these trends have led to problems in global supply [17], increasing the total cost of ownership (TCO) for data center providers. Over the last year, for example, the average DRAM DDR4 price has increased by 2.3× [1].

Our goal in this work is to reduce the DRAM footprint of MyRocks, while maintaining comparable mean latency, 99th percentile latency, and queries-per-second (QPS). The first obvious step would be to simply reduce the DRAM capacity on a MyRocks server configuration. Unfortunately, reducing the amount of DRAM degrades the performance of MyRocks. To demonstrate this point, Figure 1 shows that when reducing the DRAM cache in MyRocks from 96 GB to 16 GB, the mean latency increases by 2×, and P99 latency increases by 2.2×.

NVM offers the potential to reduce the dependence of systems like MyRocks on DRAM. NVM comes in two forms: a more expensive byte-addressable form, and a less expensive block device. Since our goal is to reduce total cost of ownership, we use NVM as a block device. Table 1 provides a breakdown of key DRAM, NVM, and Triple-Level Cell (TLC) flash characteristcs in a typical data center server. An NVM block device offers 10× faster reads and has 5× better durability than flash [3]. Nevertheless, NVM is still more than 100× slower than DRAM. To make up for that, NVM can be used to reduce the number of accesses to the slower flash layer, by significantly increasing the overall cache size. To give a sense of the price difference between NVM and DRAM, consider that on October 23, 2017, Amazon.com lists the price for an Intel Optane NVM device with 16 GB at $39, compared to $170 for 16 GB of DDR4 DRAM 2400MHz.

However, there are several unique challenges when replacing DRAM with an NVM device. First, NVM suffers from significantly lower read bandwidth than DRAM, and its bandwidth heavily depends on the block size. Therefore, simply substituting DRAM with

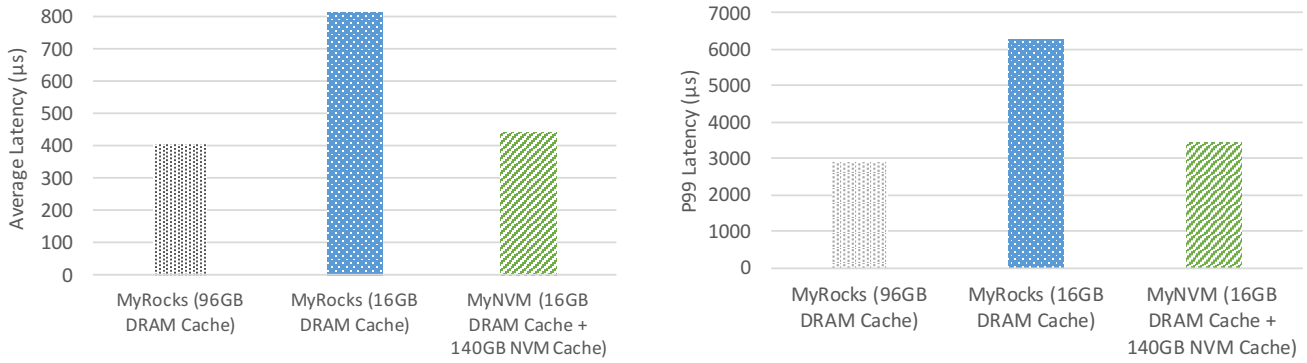


Figure 1: Average and P99 latencies for different cache sizes in MyRocks, compared with MyNVM, using real production workloads.

NVM will not achieve the desired performance due to read bandwidth constraints. In addition, significantly reducing the DRAM footprint results in a smaller amount of data that can be cached for fast DRAM access. This requires a redesign of the indices used to lookup objects in RocksDB. Second, using smaller data blocks reduces their compression ratio, thus increasing the total database size on flash. Third, unlike DRAM, NVM has write durability constraints. If it is used as a DRAM replacement without modification, it will wear out too quickly. Fourth, because NVM has a very low latency relative to other block devices, the operating system interrupt overhead becomes significant, adding a 20% average latency overhead.

We present MyNVM, a system built on top of MyRocks, that significantly reduces the DRAM cache using a second-layer NVM cache. Our contributions include several novel design choices that address the problems arising from adopting NVM in a data center setting:

(1) NVM bandwidth and index footprint: Partitioning the database index enables the caching of fine grained index blocks. Thus, the amount of cache consumed by the index blocks is reduced, and more DRAM space is left for caching data blocks. Consequently, the read bandwidth of the NVM is also reduced. When writing to NVM, compressed blocks are grouped together in the same NVM page (when possible) and aligned with the device pages, to further reduce NVM read bandwidth.
(2) Database size: When using smaller block sizes, compression becomes less efficient. To improve the compression ratio we utilize a dictionary compression.
(3) Write durability: Admission control to NVM only permits writing objects to NVM that will have a high hit rate, thus reducing the total number of writes to NVM.
(4) Interrupt latency: Hybrid polling (where the kernel sleeps for most of the polling cycle) can reduce the I/O latency overhead by 20% in certain server configurations, while minimizing CPU consumption.

We implemented MyNVM and deployed it in a MyRocks production environment in Facebook, and were able to reduce the DRAM cache by 6×. Our implementation achieves a mean latency and P99 latency of 443 μs and 3439 μs, respectively. As shown in Figure 1, both its mean and P99 latencies are 45% lower than the MyRocks server with a low amount of DRAM. Compared to the MyRocks with the high amount of DRAM, MyNVM's average latency is only 10% higher, and its P99 latency is 20% higher. Its QPS is 50% higher than the MyRocks server with a small amount of DRAM, and only 8% lower than MyRocks with the large DRAM cache.

Figure 2: The format of a Sorted String Table (SST).

2 BACKGROUND

To give context for our system, we provide a primer on RocksDB and MyRocks, and discuss the performance and durability challenges of NVM technology.

2.1 RocksDB and MyRocks

RocksDB is a popular log-structured merge tree (LSM) key-value store for SSD. Since many applications still rely on SQL backends, MyRocks is a system that adapts RocksDB to support MySQL queries, and is used as the primary database backend at Facebook [23].

In RocksDB [5], data is always written to DRAM, where it is stored in Memtables, which are in-memory data structures based on Skiplists. Once a Memtable becomes full, its data is written into a Sorted String Table (SST), which is depicted in Figure 2. An SST is a file on disk that contains sorted variable-sized key-value pairs that are partitioned into a sequence of data blocks. In addition to data blocks, the SST stores meta blocks, which contain Bloom filters and metadata of the file (e.g. properties such as data size, index size, number of entries etc). Bloom filters are space-efficient probabilistic bit arrays that are used to determine whether the SST file may contain an arbitrary key. At the end of the file, the SST stores one index block. The key of an index entry is the last key

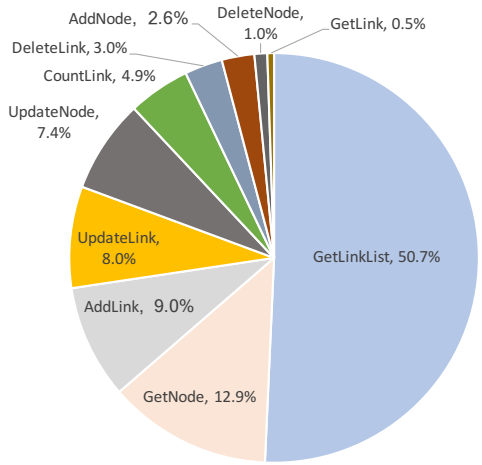


Figure 3: The distribution of requests in the LinkBench benchmark.

in the corresponding data block, and the value is the offset of that block in the file.

When executing a read query, RocksDB first checks whether its key exists in the Bloom filters. If it does, it will try to read the result of the query. It will always first check the in-memory data structures before reading from disk. In addition to storing memtables in DRAM, RocksDB also caches hot data blocks, as well as index and filter blocks.

SST files are stored in flash in multiple levels, called L0, L1, and so forth. L0 contains files that were recently flushed from the Memtable. Each level after L0 is sorted by key. To find a key, a binary search is first done over the start key of all SST files to identify which file contains the key. Only then, an additional binary search is done inside the file (using its index block) to locate the exact position of the key. Each level has a target size (e.g., 300 MB, 3 GB, 30 GB), and once it reaches its target size, a compaction is triggered. During compaction, at least one file is picked and merged with its overlapping range in the next level.

2.2 LinkBench

LinkBench is an open-source database benchmark that is based on traces from production databases that store the "social graph" data at Facebook [6]. The social graph is composed of nodes and links (the edges in the graph). The operations used to access the database include point reads, range queries (e.g. list of edges leading to a node), count queries, and simple create, delete, and update operations. Figure 3 depicts the operation distribution we used when running LinkBench, which reflect the same distribution as data center production workloads.

The LinkBench benchmark operates in two phases. The first phase populates the database by generating and loading a synthetic social graph. The graph is designed to have similar properties to the social graph, so that the number of edges from each node follows a power-law distribution. In the second phase, LinkBench runs the benchmark with a generated workload of database queries. It spawns many request threads (we use 20), which make concurrent requests to the database. Each thread generates a series of database operations that mimics the production workload. For example, the access patterns create a similar pattern of hot (frequently accessed) and cold nodes in the graph. The nodes and edges have a log-normal payload size distribution with a median of 128 bytes for the nodes and 8 bytes for the edges.

We utilize LinkBench to drive our design section, then evaluate MyNVM using real production traffic in the evaluation section.

2.3 NVM

NVM is a new persistent storage technology with the potential of replacing DRAM both in data center and consumer use cases. NVM can be used in two form factors: the DIMM form factor, which is byte-addressable, or alternatively as a block device. Unlike much of the previous work on NVM that focused on the DIMM use case [25, 27, 28], in this work, we use NVM solely as a block device. For our use case, we preferred the block-device form factor, given its lower cost.

To better understand the performance characteristics of NVM, we ran Fio 2.19 [2], a widely used I/O workload generator, on an NVM device. We ran the Fio workloads with 4 concurrent jobs with different queue depths (the queue depths represent the number of outstanding I/O requests to the device), using the Libaio I/O engine. We compared the latency and bandwidth of an NVM device with a capacity of 375 GB, to a 3D Triple-Level Cell (TLC) flash device with a capacity of 1.5 TB. TLC devices are currently the prevailing flash technology used in datacenters, because they can store 3 bits per cell, and are more efficient than devices of lower density such as Multi-level Cell (MLC) flash device (storing only 2 bits per cell).

2.3.1 ***Latency.*** Figure 4 presents the mean and P99 read latencies under 100% reads, and 70% reads and 30% writes workloads with random accesses, while Figure 5 presents the results for write latencies. For a workload that solely consists of 4 KB read operations, NVM is up to 10× faster than flash for low queue depths.

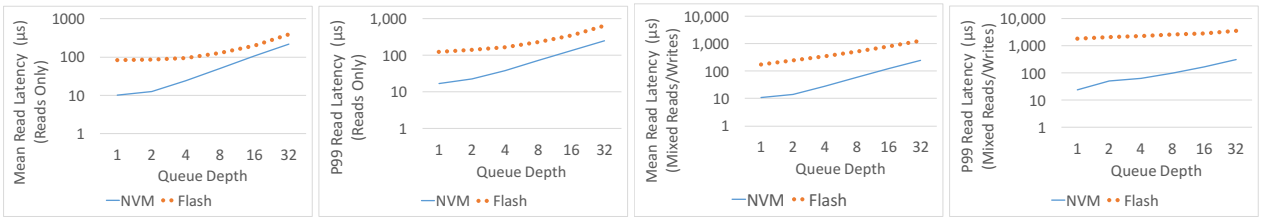
In the more realistic workload of mixed reads and writes, flash suffers from read/write interference that induces a high performance penalty. This penalty is negligible for NVM, which makes NVM's speedup over flash even more significant.

Figure 7(a) and Figure 7(b) show the mean and P99 latencies, respectively, of reading 4 KB blocks compared to reading larger blocks. Using read block sizes that are larger than 4 KB is not beneficial, as the mean and P99 latencies increase almost proportionally.

2.3.2 ***Bandwidth.*** Figure 6(a) shows the bandwidths of NVM and flash for a workload of 100% reads. Since the capacity of NVM devices is usually lower than flash, the bandwidth per capacity is much higher than flash. For a workload of mixed reads and writes, the total bandwidth of both flash and NVM is affected, yet flash deteriorates more significantly, as shown in Figure 6(b).

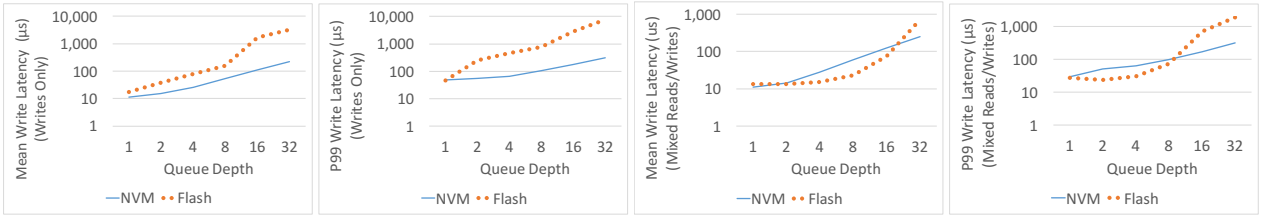
To measure the maximum NVM bandwidth for different read and write ratios, we ran Fio with a constant queue depth of 32 under different ratios. Figure 6(c) depicts the read bandwidth, write bandwidth and the combined bandwidth for various ratios.

The results show that when the relative number of writes increases, the combined bandwidth decreases from 2.2 GB/s (reads only) to 1.8 GB/s for the case of 50% reads and 50% writes. This



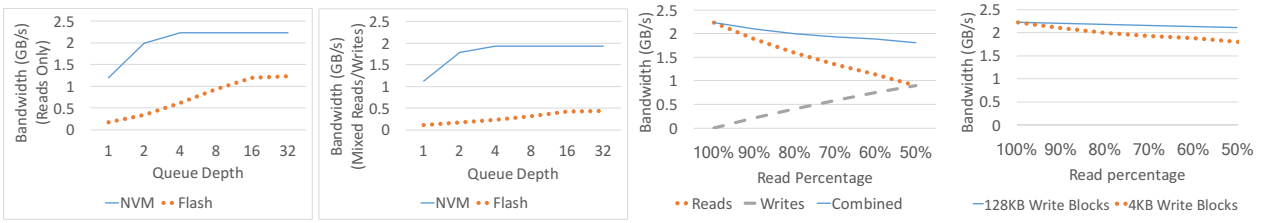
(a) (b) (c) (d)

Figure 4: Mean and P99 read latencies for a 100% reads workload, and for a workload with 70% reads and 30% writes under variable queue depths. All figures use 4 K blocks.



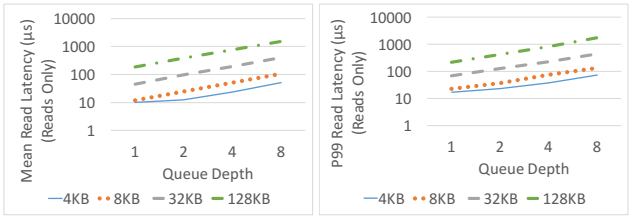
(a) (b) (c) (d)

Figure 5: Mean and P99 write latencies for a 100% writes workload, and for a workload with 70% reads and 30% writes under variable queue depths. All figures use 4 K blocks.



(a) (b) (c) (d)

Figure 6: (a) Bandwidth for a 100% read workload under different queue depths, (b) Total bandwidth for a workload with 70% reads and 30% writes under different queue depths, (c) read, write, and total bandwidth for different ratios of read and write operations, (d) total bandwidth for 4 KB read blocks with different write block sizes under variable ratios of read and write operations.



(a) (b)

Figure 7: Mean and P99 read latency for different block sizes under variable queue depths.

decrease can be mitigated by using larger write blocks, as shown in Figure 6(d).

2.3.3 ***Durability.*** Similar to flash, NVM endurance deteriorates as a function of the number of writes to the device. In general, it can tolerate about 6× more writes than a typical TLC flash device [3, 4]. Flash suffers from write amplification because it must be erased before it can be rewritten, and its erase operation has much coarser granularity than writes. Unlike flash, NVM does not suffer from write amplification, and therefore it does not deteriorate faster if it is written to randomly or in small chunks. However, since it is faster than flash, and in the case of our system it is used as a replacement for DRAM, it is likely to get written to more than flash.

3 INTUITION: THE CHALLENGE OF REPLACING DRAM WITH NVM

NVM is less expensive than DRAM on a per-byte basis, and is an order-of-magnitude faster than flash, which makes it attractive as a second level storage tier. However, there are several attributes of NVM that make it challenging when used as a drop-in replacement for DRAM, namely its higher latency, lower bandwidth, and endurance.

As we will show below, even though NVM has higher latency than DRAM, it can compensate for the lower latency if it is provisioned at a higher capacity (e.g., in our case, we use 140 GB of NVM as a replacement for 80 GB of DRAM). By that, it offers a higher cache hit rate, which results in a similar overall end-to-end latency for the entire system.

However, NVM's bandwidth is a bottleneck on the system's end-to-end performance. As we showed in §2.3, NVM's peak read bandwidth is about 2.2 GB/s. That is about 35× lower than the DRAM read bandwidth of modern servers. In addition, we use NVM as a block device. Hence, when it is used as a substitute for DRAM, the system needs to account for the fact that data can only be read from NVM at a granularity of 4 KB pages. As a result, even a small read request from the NVM causes 4 KB page to be read, and if the system naïve writes data to the device, a small amount of data might be stored on two different pages. Both of these factors further amplify the read bandwidth consumption of the system. Consequently, due to all of these factors, we found the NVM's limited read bandwidth to be the most important inhibiting factor in the adoption of NVM for key-value stores.

In addition, unlike DRAM, NVM suffers from an endurance limitation. If its cells are written to more than a certain number of times, they wear out, and the device may start to experience a higher probability of uncorrectable bit errors, which shortens its lifetime. This problem is exacerbated when using NVM as a cache. In cache environments, cold objects are frequently evicted, and replaced by newly written objects. As we will demonstrate, this results in the cache workload far exceeding the allowed number of writes, and rapidly wearing the device. Therefore, NVM cannot simply be used as a drop-in replacement for a DRAM cache, and the number of writes to it needs to be limited by the system.

Finally, since NVM has a very low latency relative to other block devices, the operating system interrupt overhead becomes significant when used with NVM as a block device. As shown in §2.3, the average end-to-end read latency of NVM is on the order of 10μs when used in low queue depths. Included in this number is an overhead of two context switches due to the operating system's interrupt handling, which are on the order of 2μs. These result in a roughly 20% extra overhead under low queue depths.

We present MyNVM, a system that addresses these challenges, by adapting the architecture of the key-value store and its cache to the unique constraints of NVM. The same principles we used in the design of MyNVM can be utilized when building other data center storage systems that today rely solely on DRAM as a cache.

4 DESIGN

The architecture of MyNVM is depicted in Figure 8. MyNVM is built on top of MyRocks. Similar to MyRocks, MyNVM utilizes DRAM as a write-through cache for frequently accessed blocks, and flash as the medium that persists the entire database. MyNVM uses NVM as a second level write-through cache, which has an order of magnitude greater capacity than DRAM. Both DRAM and NVM by default use least recently used (LRU) as their eviction policy.

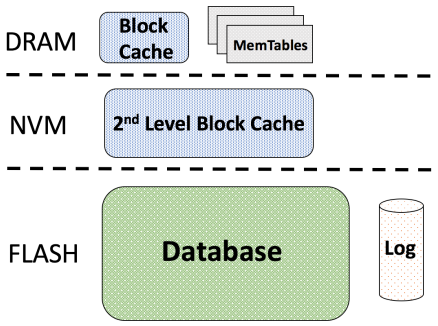


Figure 8: MyNVM's architecture. NVM is used as a second level block cache.

The design goal for MyNVM is to significantly reduce the amount of DRAM compared to a typical MyRocks server, while maintaining latency (mean and P99), as well as QPS, by leveraging NVM. In this section, we describe the challenges in replacing DRAM with NVM, including bandwidth consumption, increased database size, NVM endurance, and interrupt latency, as well as our implemented solutions in MyNVM.

A MyRocks (or MySQL) setup consists of instances. Backing up and migrating data between hosts is done in the granularity of an instance. When the instances are too large, tasks such as load balancing, and backup creation and restoration become less efficient. Thus, MyRocks servers are often configured to run multiple smaller MyRocks instances, rather than using a single large instance.

Unfortunately, LinkBench currently supports loading only a single MyRocks instance. To realize the bottlenecks and requirements of incorporating NVM with a typical MyRocks server, we run our Linkbench experiments with a single instance and extrapolate the results for 8 instances.

4.1 Satisfying NVM's Read Bandwidth

The first obstacle to replacing DRAM with NVM is NVM's limited read bandwidth. Figure 9 depicts the bandwidth consumed by MyNVM using 16 KB blocks, which is the default setting, running the LinkBench [6] workload. As the figure shows, the NVM device only provides 2.2 GB/s of maximum read bandwidth, and MyNVM requires more than double that bandwidth.

The main reason that MyNVM consumes so much read bandwidth is that in order to read an object, it has to fetch the entire data block that contains that object. Read sizes in the LinkBench workload are on the order of tens to hundreds of bytes. Hence, every time MyNVM needs to read an object of 100 B, it would need to read about 160× more data from the NVM, because data blocks are 16 KB.

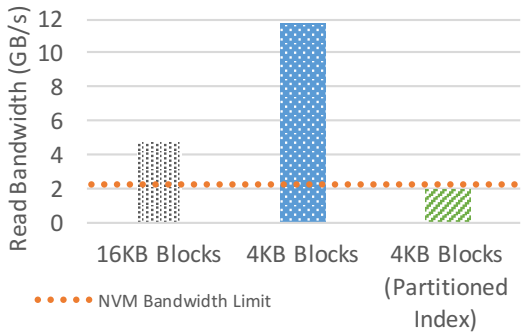


Figure 9: Read bandwidth consumed by MyNVM using 16 KB blocks, 4 KB blocks, and 4 KB blocks with partitioned index, running the LinkBench [6] workload.

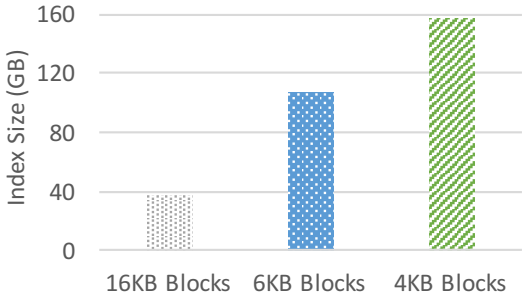


Figure 10: Total index size for different block sizes in MyNVM.

4.1.1 ***Reducing Block Size.*** Therefore, a straightforward solution might be to simply reduce the data block size, so that each read consumes less bandwidth. However, instead of decreasing the bandwidth consumption, when we reduce the block size from 16 KB to 4 KB, the bandwidth consumption more than doubles (see Figure 9).

The reason the bandwidth consumption increases is because as the block size decreases, the DRAM footprint of the index blocks becomes significant. Figure 10 depicts the size of the index with different block sizes in MyNVM. Moving from 16 KB blocks to 4 KB quadruples the size of the index. As shown by Figure 11, this results in a lower DRAM hit rate, since the index takes up more space.

The index blocks contain one entry per data block and service an entire SST file, which accommodates tens of thousands of data blocks, some of which might be relatively cold. Therefore, to address this problem, we partition the index into smaller blocks, and add an additional top-level index where each entry points to a partitioned index block. Thus, instead of having a single index block per SST, the index is stored in multiple smaller blocks. Figure 12 illustrates the partitioned index.

When reading an index block from disk, only the top-level index is loaded to memory. Based on the lookup, only the relevant index partitions are read and cached in DRAM. Hence, even with 4 KB blocks, partitioning increases the hit rate to a similar level as 16 KB

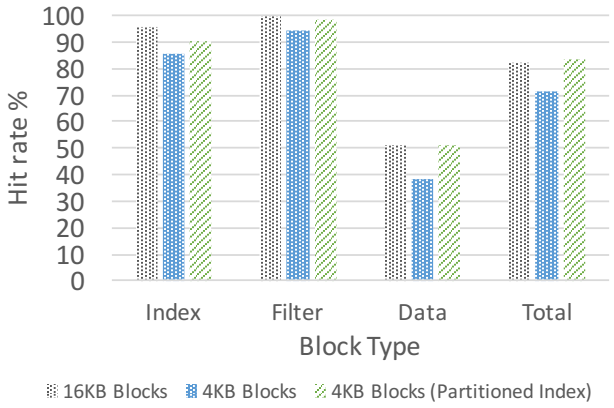


Figure 11: DRAM hit rate of different block types when using 16KB blocks, 4KB blocks, and 4KB blocks with partitioned index

Figure 12: Non-partitioned and paritioned index structures. The partitioned scheme uses a top-level index, which points to lower-level indices.

(see Figure 11), and reduces overall read bandwidth consumption to less than 2 GB, which is at the limit of the NVM device.

However, even 2 GB bandwidth consumption is almost at the full capacity of the device. Consequently, we need to find another way to reduce bandwidth consumption without affecting performance.

4.1.2 ***Aligning Blocks with Physical Pages.*** In RocksDB, data blocks stored in levels L1 and higher are compressed by default. Since we reduced the block size in MyNVM, a 4 KB block on average would consume less than 4 KB when it is compressed and written to a physical page on NVM. The remainder of the page would be used for other blocks. Consequently, some compressed blocks are spread over two pages, and require double the read bandwidth to be read.

Thus, instead of using 4 KB blocks, MyNVM uses 6 KB blocks, which are compressed on average to about 4 KB pages when stored on flash or NVM in layers L1 and above. Since the block size is higher, this also reduces the bandwidth further because the index size is reduced (Figure 10).

However, even with 6 KB blocks, some blocks may be compressed to less than 4 KB (e.g., as low as 1 KB), and some may not be compressed at all, and would be spread over two physical 4 KB pages. Figure 13 shows the percentage of logical 6 KB that are spread over two pages.

To address this problem, when trying to pack multiple compressed blocks into a single page, we zero-pad the end of the page if the next compressed block cannot fully fit into the same page. Figure 14 depicts MyNVM's block alignment. Figure 13 shows that zero padding the blocks reduces the number of blocks that are spread

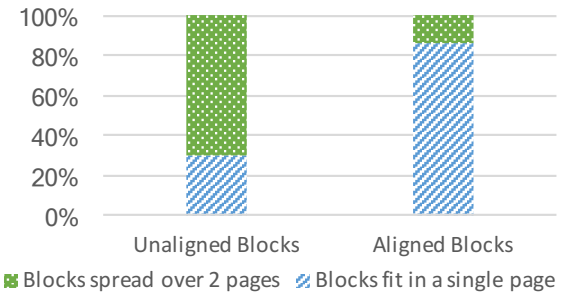


Figure 13: The percentage of blocks that are spread over two physical 4 KB pages or more. "Unaligned blocks" is the case when we do not use zero-padding at the end of the physical page, while "aligned blocks" is with zero-padding.

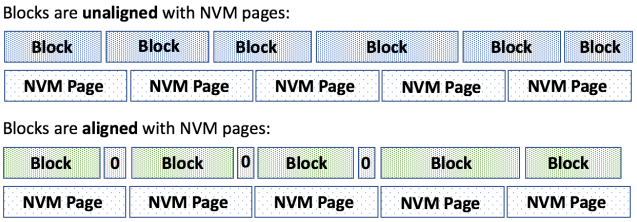


Figure 14: An illustration of blocks that are unaligned with the NVM pages, and of blocks that are aligned with the NVM pages.

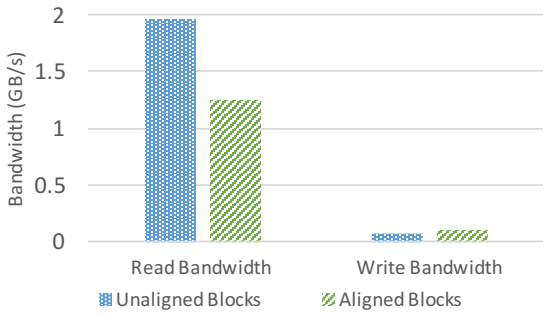


Figure 15: Read and write bandwidth for unaligned and aligned blocks.

over two pages by about 5×. Note that even with zero-padding, some logical blocks are spread over two pages, because they cannot be compressed down to 4 KB.

Figure 15 demonstrates the effect of aligning the blocks on read and write bandwidth. It shows that aligning the blocks reduces the read bandwidth by about 30% to 1.25 GB/s, because on average it requires reading fewer pages. Aligning blocks very slightly increases write bandwidth consumption, because it requires writing more pages for some blocks (due to zero-padding). In addition, the effective capacity of the NVM cache is a little smaller, because zeros occupy some of the space. This reduces the NVM cache hitrate by

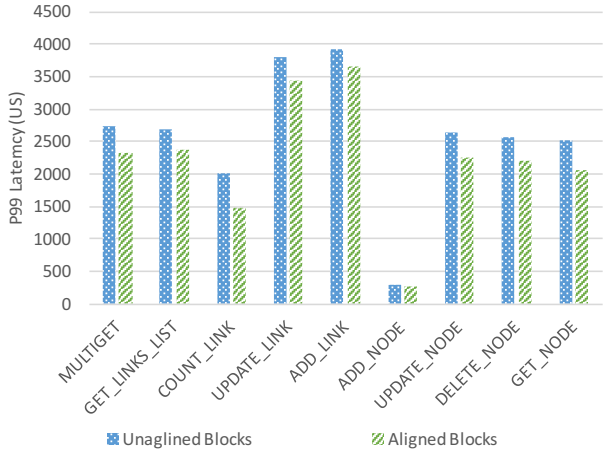


Figure 16: P99 latencies with unaligned and aligned blocks.

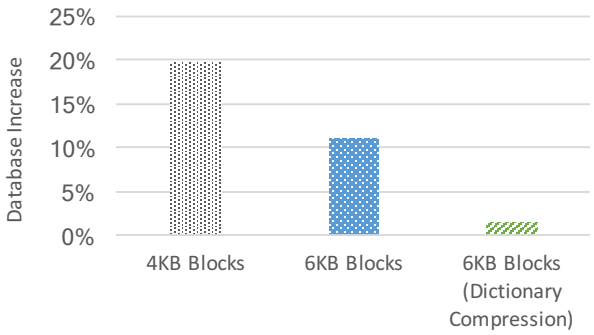


Figure 17: Database increase when using 4 KB and 6 KB logical blocks, compared with database size of 16KB blocks

1%-2%, which is compensated by reading less pages in the case of aligned blocks.

For the same reason, block alignment also reduces the P99 latency. Figure 16 shows that the P99 latency with the LinkBench workload is improved with block alignment.

4.2 Database Size

A negative side-effect of using smaller block sizes is that compression becomes less efficient, because compression is applied on each data block individually. For each compressed block, RocksDB starts from an empty dictionary, and constructs the dictionary during a single pass over the data. Therefore, the space consumed by the database on the flash increases.

Figure 17 depicts the increase in database size. Logical blocks of 4 KB increase the database size by almost 20%. Using logical blocks of 6 KB still has an overhead of more than 11%.

To address this overhead, MyNVM preloads the dictionary from data that is uniformly sampled from multiple blocks. This increases the compression ratio when the same patterns are repeated across

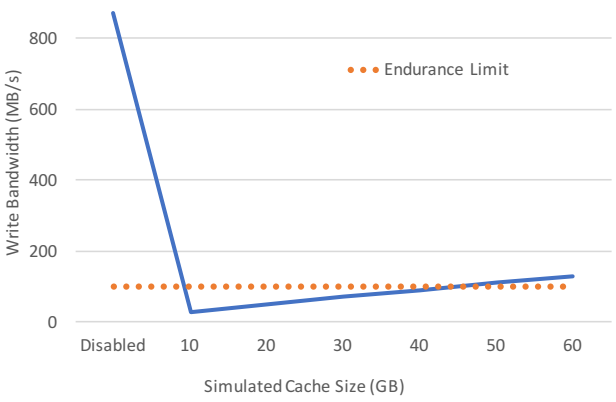


Figure 18: Write bandwidth to NVM for different simulated cache sizes, per instance. Disabled means the NVM is used without an admission policy. The endurance limit is derived from the Device Writes Per Day specification.

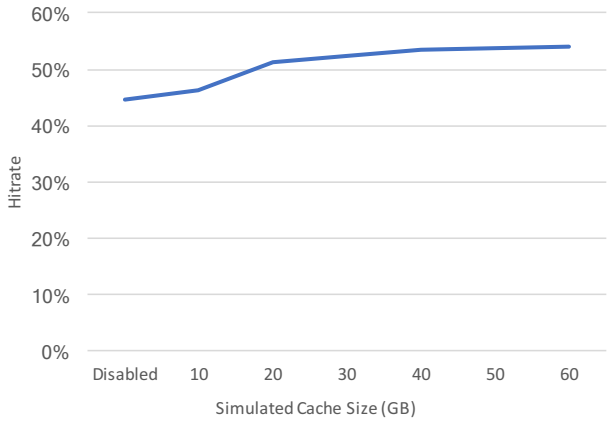


Figure 19: Hit rates for different simulated cache sizes per instance. Disabled means the NVM is used without an admission policy.

multiple blocks. Figure 17 shows that with dictionary compression, the size overhead decreases to only 1.5%.

4.3 NVM's Durability Constraint

While NVM improves flash durability, it is still constrained by the number of writes it can tolerate before it experiences uncorrectable bit errors.

Figure 18 depicts the write bandwidth to NVM under the LinkBench workload. It shows that if NVM is simply used as a second level cache, its write bandwidth would significantly exceed the endurance limit of the device (i.e., its DWPD limit), which would cause it to rapidly wear out.

Therefore, instead of naïvely using NVM as a second level cache, we design an admission control policy that only stores blocks in NVM that are not likely to get quickly evicted. To do so, we maintain an LRU list in DRAM that represents the size of the NVM and only stores the keys of the blocks that are being accessed. This simulates a cache without actually caching the values of the blocks.

When a block is loaded from flash, we only cache it in NVM if it has been recently accessed, and therefore already appears in the simulated cache LRU. Figure 19 demonstrates the hit rate increases as a function of the size of the simulated cache. As the simulated cache grows, MyNVM can more accurately predict which objects are worth caching in the NVM layer. Beyond a size of 30-40 GB, increasing the size of the simulated cache offers diminishing returns.

Figure 18 shows the write bandwidth after we apply admission control to NVM, as a function of the size of the simulated cache. A large simulated cache will admit more objects to NVM, which will increase the hit rate, but also suffer from a higher write bandwidth. In the case of MyNVM, a simulated cache that represents 40 GB of NVM data per instance achieves a sufficiently low write bandwidth to accommodate the endurance limitation of the NVM device.

4.4 Interrupt Latency

Due to NVM's low latency as a block device, the overhead of interrupt handling becomes significant. In order to serve a block I/O request, the operating system needs to submit the request asynchronously. When that is done, the application sleeps until the request has been completed on the hardware side. This will either trigger a context switch, if the CPU scheduler has other tasks to run on that CPU, or a switch to a lower power state if the CPU goes idle. Both of those outcomes have a cost associated with them. Particularly, the enter and exit latencies for lower power states have a non trivial cost. Once the device completes the request, it raises an interrupt which wakes up the task that submitted the I/O. Figure 20(a) illustrates this process.

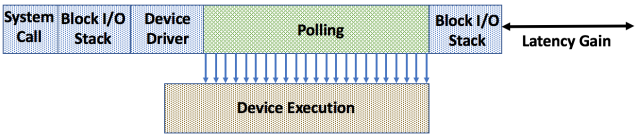
Replacing interrupts with polling is a way to reduce this overhead. With polling, the operating system continuously checks whether the device has completed the request. As soon as the device is ready, the operating system is able to access the device, either avoiding a context switch or the expensive power state enter/exit latencies. This process is described in Figure 20(b). By reducing the overall latency, the effective bandwidth of the NVM device increases, and thus also its overall QPS.

To explore the potential of polling, we ran Fio with the pvsync2 I/O engine, which provide polling support. Figure 21(a) shows the increase in available NVM bandwidth when using polling compared with interrupts. However, continuously polling brings the CPU consumption of an entire core to 100%, as shown in Figure 21(b).

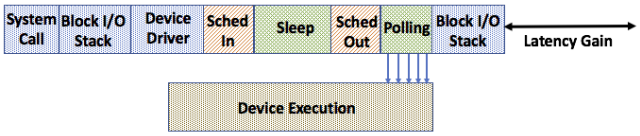
To overcome this, we utilize a hybrid polling policy, where the process sleeps for a fixed time threshold after the I/O was issued until it starts polling, as illustrated in Figure 20(c). Figure 21(a) and Figure 21(b) present the bandwidth and CPU consumption of the hybrid polling policy with a threshold of 5 μs.

While this policy significantly reduces CPU consumption, the CPU consumption still increases with the number of threads because the polling threshold is static. To overcome this, we propose

(a) The I/O latency when using interrupts



(b) The I/O latency when using polling



(c) The I/O latency when using hybrid polling

Figure 20: Diagrams of the I/O latencies as perceived by the application, using interrupts, polling, and hybrid polling.

a dynamic hybrid polling technique. Instead of using a preconfigured time threshold, the kernel collects the mean latencies of past I/O and triggers polling based on that. Hence, the polling begins only after a certain portion threshold of the past mean I/O latency is reached. By default, the kernel will sleep for 50% of the mean completion time for an I/O of the given size. This means that the kernel will start polling only after a duration that is half the mean latency of past requests. We added a patch that makes this parameter configurable, allowing us to experiment with settings other than 50% of the mean completion time. Figure 21(c) and Figure 21(d) depict the bandwidth and CPU consumption for different latency percentage thresholds.

While polling can provide a significant boost to the NVM bandwidth as long as the number of threads is small, in the case of 8 threads (or more) the benefits diminish. Further experimentation is needed, but we hypothesize that this is largely due to several primary effects. One is that as more threads are accessing the device, the individual I/O latencies start to increase, thus diminishing the benefits of polling. Secondly, as more threads are running in the system, there's a higher chance that a CPU will have pending work outside of the thread submitting I/O. This aborts the polling done by the kernel, as it prefers to schedule other work instead of polling, if other work is available. Thirdly, as more threads are polling for IO completion, the polling overhead becomes larger. One way to fix this in the kernel would be to offload polling to dedicated kernel threads, instead of letting the application itself poll the NVMe completion queue. The application directed poll yields lower latencies for synchronous (or low queue depth) IO, where offloading work to another thread would add latency. However, for higher queue depth and pipelined IO, benefits could be seen by having dedicated kernel threads handle the polling. Lastly, the Linux kernel utilizes all NVMe submissions/completion queue pairs for polling. As the number of polling threads increase, so does the number of queues that need to be polled. A related optimization would be to create dedicated pollable submission queues, reducing the number of queues that need to be checked for completions. This in turn could reduce the required threads needed to reap completions by polling. An improved polling mechanism in the kernel could remove many of these limitations. Until that is available, we decided to currently not integrate polling in our production implementation of MyNVM, but plan to incorporate it in future work.

4.5 Applicability to Fast Flash Devices

Similar to NVM devices, fast flash devices promise lower latencies and better endurance compared with typical flash devices, while still suffering from challenging throughput and endurance limitiations. Some of the solutions presented in MyNVM may be applied also to fast flash devices. However, since these devices suffer from flash side-effects that do not exist in NVM, such as write amplification and read/write interference, they would require additional design considerations.

5 IMPLEMENTATION

MyNVM is implemented on top of the open-source MyRocks and RocksDB. The initial implementation has been committed to the open source repository, and we plan to add the rest of it.

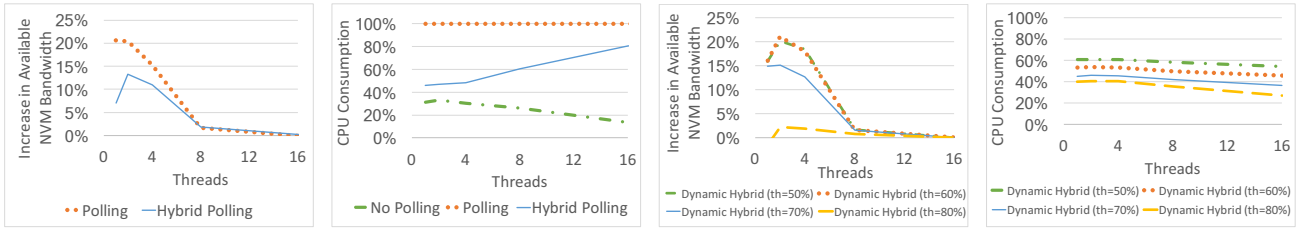
MyNVM uses DRAM as a first level write-through block cache. Since the vast majority of queries hit in DRAM, blocks are always cached to DRAM uncompressed. Otherwise, compressing and decompressing the requests on each access would add a significant CPU overhead. To enable this function, reads from and writes to the NVM and flash devices are done using direct I/O, bypassing the operating system page cache. Avoiding the operating system page cache also prevents redundant copy operations between the devices and DRAM (where the page cache acts as a proxy).

NVM serves as a second level write-through block cache. Due to its read bandwidth and endurance limit, blocks in the NVM are compressed, similar to the blocks stored in the flash.

During lookup, MyNVM first accesses the DRAM, and only in case of a miss it will look for the object in NVM. In case of a miss in the NVM, MyNVM will not always load the missed key to NVM, due to the NVM endurance limit. MyNVM's admission control policy decides which key will be loaded to NVM, by simulating a larger cache. The simulated cache is based in LRU and is updated at each query, regardless of whether the requested block resides in the block caches.

To avoid frequent writes and updates to NVM, the NVM lookup table is maintained in DRAM. In order to keep the table lightweight, it does not store the entire keys, but instead uses a hashed signature of up to 4 bytes per block. Due to the rare cases of aliasing (where two keys hash to the same signature), a block will always be validated after it is read from the NVM. If its key differs from the desired key, it is considered a miss and the block will be read from flash.

Writes to NVM are done in batches of at least 128KB, because we found that smaller write sizes degrade the achieved read bandwidth (see §2.3). Cached blocks are appended into larger NVM files with a configurable size. The eviction policy from NVM is maintained



(a) (b) (c) (d)

Figure 21: Left to right: (a) the increase in NVM read bandwidth when using polling compared to interrupts, (b) the increase in CPU when using polling and hybrid polling compared to interrupts, (c) the increase in NVM read bandwidth with dynamic hybrid polling, and (d) CPU consumption of dynamic hybrid polling with different thresholds.

in granularity of these files, which significantly reduces the in-memory overhead. The files are kept in an eviction queue, and once the NVM capacity limit is reached, the least recently used file is evicted to make room for a new file.

As we showed in §4.1, MyNVM uses a partitioned index and 6 KB blocks to reduce read bandwidth. Thus, instead of having a single block index for each SST file, the index is partitioned to multiple smaller blocks with a configurable size. We partitioned the index to 4 KB blocks, so they align with the memory page size as well as NVM page size (index blocks are never compressed). In addition, the partitioned index includes a new top-level index block, which serves as an index for the partitioned index blocks. Each one of the index blocks (including the top-level index block) is sorted. To find if a key exists in the SST file, MyNVM first performs a binary search in the top-level index block. Once the relevant partition is found, it will perform a binary search within the partition to find the key, and its corresponding data block.

In addition, we implemented block alignment, so that single blocks will be stored over the minimal number of physical NVM pages possible (see §4.1.2). This number is calculated by the following formula:

$$\lfloor \frac{Compressed\ Block\ Size - 1}{NVM\ Page\ Size} \rfloor + 1$$

Before a block is written to the NVM, its expected starting address and end address in NVM are compared, to predict the number of NVM pages that will be occupied. If this number is larger than the minimal number of required NVM pages (e.g. 1 page for 4KB block), the previous block will be padded with zeroes until the end of the page. Thus, the new block will be written at the beginning of the next page.

MyNVM uses the Zstandard compression algorithm [12] with dictionary compression. When a new SST file is written to flash (during LSM compactions), it samples the SST file with a uniform distribution (each sample is 64 bytes) to create a joint dictionary of up to 16KB (we found that larger dictionaries did not improve the compression ratios). Due to the fact that keys are sorted inside the blocks, only the delta between consecutive keys is kept in the block. Thus, rather than sampling from the keys and values of the block, the sampling is done across the raw blocks. The dictionary is stored in a dedicated block in the SST, and is used for the compression and decompression of all the SST data blocks.

In order to incorporate dynamic hybrid polling in MyNVM (see §4.4), we used the synchronous pwritev2 and preadv2 system calls for reading and writing to and from the NVM. These two system calls are a recent addition to the Linux kernel. The interface is similar to pwritev and preadv, but they take an additional flags argument. One of the supported flags is RWF_HIPRI, allowing an application to inform the kernel that the IO being submitted is of high priority. If this flag is used in conjunction with the file descriptor being opened for O_DIRECT (unbuffered) IO, the kernel will initiate polling after submitting the request, instead of putting the task to sleep. Device level settings define what kind of polling will be used, from 100% CPU busy polling, fixed delay hybrid polling, or dynamic hybrid polling. However, because of the inefficiencies experienced with more than 8 threads, we did not include it in the final system.

6 EVALUATION

This section presents a performance analysis of MyNVM under production environments. We evaluate MyNVM's mean latency, P99 latency, QPS, and CPU consumption, and compare it with MyRocks. We use dual-socket Xeon E5 processor servers with 3 different configurations: MyRocks with a large amount of DRAM cache (96 GB), small amount of DRAM cache (16 GB), and MyNVM with 16 GB of DRAM cache and 140 GB NVM. For most of this section (except for the QPS evaluation), we ran a shadow workload by capturing production MyRocks workloads of the user database (UDB) tier, and replaying them in real-time on these machines.

6.1 Mean and P99 Latency

Figure 1 shows that with MyRocks, reducing the DRAM cache from 96 GB to 16 GB, more than doubles both the mean latency and the P99 latency.

With a mean latency of 443 μs and a P99 latency of 3439 μs, MyNVM's mean and P99 latencies are both 45% lower than the MyRocks server with a small amount of DRAM. The average latency of MyNVM is 10% higher, and its P99 latency is 20% higher, than the MyRocks configuration with a 96 GB DRAM cache.

Figure 22 and Figure 23 provide the mean latency and P99 latency over time, with intervals of 5M queries, for a time period of 24 hours.

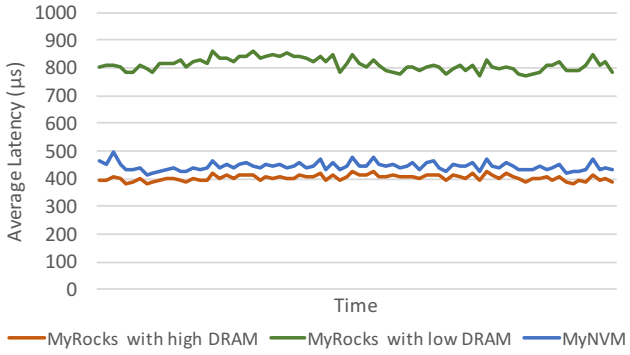


Figure 22: Mean latencies of MyRocks with 96 GB of DRAM cache, compared to MyRocks with 16 GB of DRAM cache, and MyNVM with 16 GB of DRAM cache and 140GB of NVM, over a time period of 24 hours.

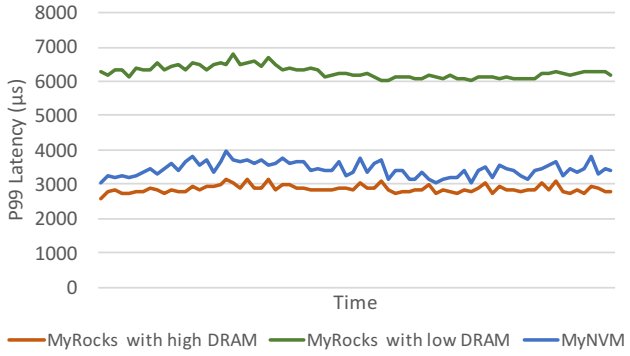


Figure 23: P99 latencies of MyRocks with 96 GB of DRAM cache, compared to MyRocks with 16 GB of DRAM cache, and MyNVM with 16 GB of DRAM cache and 140GB of NVM, over a time period of 24 hours.

6.2 QPS

To evaluate MyNVM's QPS, we could not use the shadow workload, because it does not drive the maximum amount of traffic MyNVM can handle. Therefore, we use the LinkBench workload.

Figure 24 presents the NVM hit rate and MyNVM's QPS as a function of the size of the NVM cache. The graph shows that beyond 140 GB NVM, the NVM hit rate has diminishing returns, and consequently MyNVM's QPS also offers diminishing returns. This led us to to choose 140 GB NVM as the default configuration for MyNVM.

Figure 25 compares the QPS of MyRocks with different cache sizes and MyNVM. The QPS of MyNVM is 50% higher than the MyRocks server with a small amount of DRAM, and only 8% lower than the server with the large amount of DRAM cache.

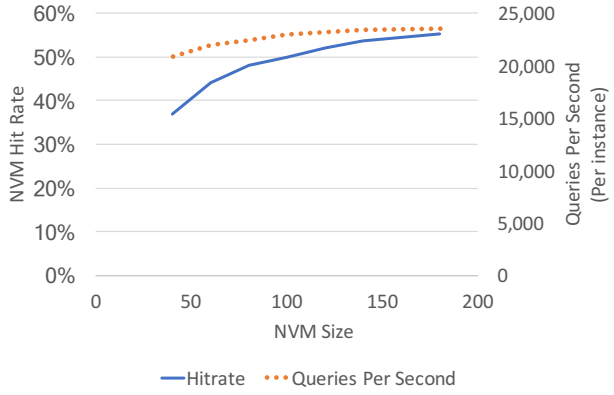


Figure 24: NVM hit rate and QPS in MyNVM as a function of NVM size.

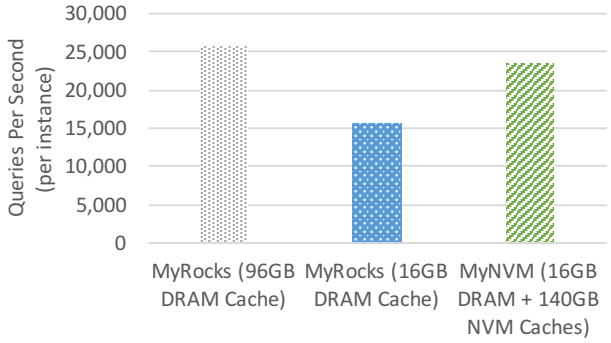


Figure 25: Queries per second (QPS) for different cache sizes in MyRocks, compared with MyNVM.

6.3 CPU Consumption

Figure 26 depicts the CPU consumption of MyRocks with a large amount of DRAM cache, small amount of DRAM cache (16 GB) and MyNVM, over a time period of 24 hours (with 15 minute intervals), running the shadow production traffic. The differences in CPU consumption trends over time are due to traffic fluctuations during the day.

On average, reducing the DRAM cache in MyRocks from 96 GB to 16 GB increases the CPU consumption from 18.4% to 28.6%. The first reason for this difference is that with smaller DRAM, fewer requests hit in the uncompressed DRAM cache, and instead are loaded from flash. Thus, more blocks have to be decompressed. In addition, for the smaller DRAM configuration, the CPU spends more time waiting for outstanding requests to the flash. Figure 27 depicts the I/O-Wait percentage over 24 hours (using 15 minute intervals), which represents the portion of the time in which the CPU is waiting for storage I/O to complete. On average, it increases from 3.7% to 7.8% when switching to MyRocks with smaller amount of DRAM.

MyNVM, on the other hand, has an average I/O-Wait of only 2.7%. Because its overall cache size is larger, it triggers fewer accessess

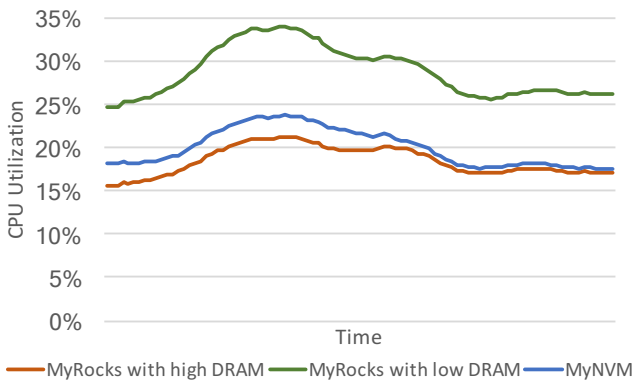


Figure 26: CPU consumption over 24 hours, of MyRocks with 96 GB of DRAM cache, compared to MyRocks with 16 GB of DRAM cache, and MyNVM with 16 GB of DRAM cache and 140 GB of NVM, over a time period of 24 hours.

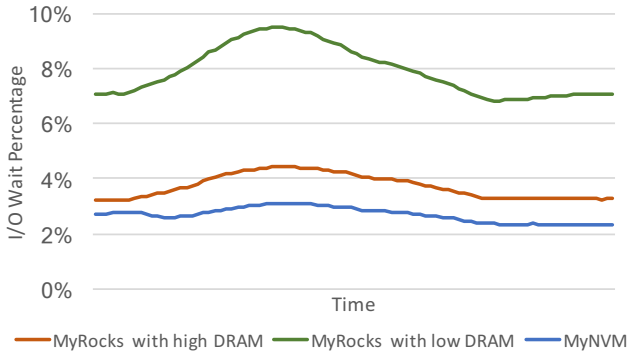


Figure 27: I/O-Wait percentage over 24 hours, of MyRocks with 96 GB of DRAM cache, compared to MyRocks with 16 GB of DRAM cache, and MyNVM with 16 GB of DRAM cache and 140 GB of NVM, over a time period of 24 hours.

to the flash compared with MyRocks (in both low and high DRAM configurations), and NVM accesses are faster. In addition, MyNVM uses smaller blocks of 6 KB, which take less time to compress or decompress compared with 16 KB blocks. Consequently, MyNVM's CPU consumption is lower than the low DRAM MyRocks, with an average of 20%. It still has higher CPU consumption than the high-DRAM MyRocks because fewer requests hit in the uncompressed DRAM, and instead are loaded from the NVM cache, which is compressed.

7 RELATED WORK

The related work belongs to two main categories: prior work on NVM in the context of key-value stores, and systems that influenced the design of MyNVM.

7.1 NVM for Key-value Stores

There are several previous projects that adapt persistent key-value stores to NVM. However, all prior work has relied on NVM emulation and was not implemented on real NVM hardware. Therefore, prior work does not address practical concerns of using NVM as a replacement for DRAM, such as dealing with NVM bandwidth limitations. In addition, prior work assumes byte-addressable NVM, and our work is focused on block-level NVM due to its cost. To the best of our knowledge, this is the first implementation of a key-value store on NVM that deals with its practical performance implications.

CDDS [27], FPTree [25] and HiKV [28] introduce B-Tree across DRAM and NVM, which minimizes writes to NVM. Echo [7] implements key-value functionality using hash-tables. All of these systems rely on emulations or simulations of NVM. In addition, unlike MyNVM, these systems do not rely on flash as an underlying storage layer, and are therefore much more expensive per bit.

Unlike these systems, NVStore [31] does not rely on emulations, but on a real evaluation. However, its evaluation is based on NVDIMM, which is not NVM, but rather a hybrid DRAM and flash system, which persists data from DRAM to flash upon power failure.

There has been additional work on indices that reduce writes to byte-addressable NVM, to reduce its wear [11, 14, 18, 32]. Prior research has also explored using using differential logging [10, 20, 21], to only log modified bytes to avoid excessive writes in NVM. Other systems utilize atomic writes for metadata, and write-ahead logging and copy-on-write for data [13, 15, 29].

7.2 Related Systems

Several aspects of our design were inspired by prior work on non-NVM key-value stores and polling.

Prior systems use admission control to control access to a slower second tier cache layer. Similar to how MyNVM controls accesses to NVM, Flashield [16] applies an admission control to reduce the number of writes to flash, and protect it from wearing out. In Flashield, objects that have not been read multiple times are only cached in DRAM, and do not get cached in flash. AdaptSize [8] applies admission control to decide which objects get written to a CDN based on their size and prior access patterns. In future work we plan to explore more sophisticated admission control policies to NVM, which incorporate features like the number of prior access and object size.

Polling has been used in prior work [26, 30] as a potential way to deliver higher performance for fast storage devices. However, there is no prior research on using polling in conjunction with NVM, and there is no work in addressing the resulting high CPU consumption, which makes its adoption impractical for many use cases. We introduced a dynamic hybrid polling mechanism, which reduces the CPU consumption significantly, and evaluated it on top of an NVM device.

8 CONCLUSIONS

To the best of our knowledge, this is the first study exploring the usage of NVM in a production data center environment. We presented MyNVM, a system built on top of MyRocks, which utilizes

NVM to significantly reduce the consumption of DRAM, while maintaining comparable latencies and QPS. We introduced several novel solutions to the challenges of incorporating NVM, including using small block sizes with partitioned index, aligning blocks with physical NVM pages, utilizing dictionary compression, leveraging admission control to the NVM, and reducing the interrupt latency overhead. Our results indicate that while it faces some unique challenges, such as bandwidth and endurance, NVM is a potentially lower-cost alternative to DRAM.

NVM can be utilized in many other data center use cases beyond the one described in this paper. For example, since key-value caches, such as memcached and Redis, are typically accessed over the network, their data can be stored on NVM rather than DRAM without incurring a large performance cost. Furthermore, since NVM is persistent, a node does not need to be warmed up in case it reboots. In addition, NVM can be deployed to augment DRAM in a variety of other databases.

9 ACKNOWLEDGMENTS

We thank Kumar Sundararajan, Yashar Bayani, David Brooks, Andrew Kryczka, Banit Agrawal, and Abhishek Dhanotia for their valuable assistance and suggestions. We also thank our shepherd, Dejan Kostic, and the anonymous reviewers for their thoughtful feedback.

REFERENCES

[1] Dram prices continue to climb. https://epsnews.com/2017/08/18/dram-prices-continue-climb/.
[2] Flexible I/O tester. https://github.com/axboe/fio.
[3] Intel Optane DC p4800x specifications. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html.
[4] Introducing the Samsung PM1725a NVMe SSD. http://www.samsung.com/semiconductor/insights/tech-leadership/brochure-samsung-pm1725a-nvme-ssd/.
[5] RocksDB wiki. github.com/facebook/rocksdb/wiki//.
[6] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM.
[7] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 4:1–4:8, New York, NY, USA, 2013. ACM.
[8] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.
[9] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013.
[10] J. Chen, Q. Wei, C. Chen, and L. Wu. FSMAC: A file system metadata accelerator with non-volatile memory. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–11. IEEE, 2013.
[11] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, pages 21–31. www.cidrdb.org, 2011.
[12] Y. COLLET and C. TURNER. Smaller and faster data compression with zstandard, 2016, 2016.
[13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.
[14] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 1:1–1:9, New York, NY, USA, 2015. ACM.
[15] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.
[16] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a key-value cache that minimizes writes to flash. *CoRR*, abs/1702.02588, 2017.
[17] D. Exchange. DRAM supply to remain tight with its annual bit growth for 2018 forecast at just 19.6 www.dramexchange.com.
[18] W. Hu, G. Li, J. Ni, D. Sun, and K.-L. Tan. B-tree: A predictive B-tree for reducing writes on phase change memory. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2368–2381, 2014.
[19] U. Kang, H.-s. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi. Co-architecting controllers and dram to enhance dram process scaling. In *The memory forum*, pages 1–4, 2014.
[20] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in write-ahead logging. *SIGPLAN Not.*, 51(4):385–398, Mar. 2016.
[21] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn. Shortcut-JFS: A write efficient journaling file system for phase change memory. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–6. IEEE, 2012.
[22] S.-H. Lee. Technology scaling challenges and opportunities of memory devices. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 1–1. IEEE, 2016.
[23] Y. Matsunobu. Myrocks: A space and write-optimized MySQL database. code. facebook.com/posts/190251048047090/.
[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013.
[25] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.
[26] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom. OS i/o path optimizations for flash solid-state drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 483–488, Philadelphia, PA, 2014. USENIX Association.
[27] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.
[28] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017. USENIX Association.
[29] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.
[30] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.
[31] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, 2015. USENIX Association.
[32] P. Zuo and Y. Hua. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33st Symposium on Mass Storage Systems and Technologies, MSST*, volume 17, pages 1–10, 2017.