Assessing Non-volatile Memory in Modern Heterogeneous Storage Landscape using a Write-optimized Storage Stack

Sajad Karim[1], Johannes Wünsche[2], David Broneske[1], Michael Kuhn[2] and Gunter Saake[1]

[1]*Databases and Software Engineering, Otto von Guericke University Magdeburg*
[2]*Parallel Computing and I/O, Otto von Guericke University Magdeburg*

Abstract

Non-volatile memory (NVM), or persistent memory, is a promising and emerging storage technology that has not only disrupted the typical long-established memory hierarchy but also invalidated the proclaimed programming paradigm used in traditional database management systems and file systems. It bridges the gap between primary and secondary storage and, hence, shares the characteristics of both categories. However, currently, there exists no common storage engine built particularly to study the characteristics of the modern storage landscape, which has become more heterogeneous after NVM. Therefore, a general-purpose storage engine, Haura, is utilized to study the benefits of the modern storage landscape. In this work, NVM is integrated into the storage stack of Haura and studied the patterns for modern storage devices involved and their impact on the performance of Haura. Our work shows, NVM performs best under sequential workloads, but random access is better with larger block sizes. Furthermore, the block size has a significant impact on the performance of storage devices, with smaller block sizes favoring NVM and larger block sizes favoring NVMe-supported devices.

1. Introduction

According to Kazemie [1], the volume of data in 2011 was around 1.8 Zettabytes, and its volume doubles approximately every two years. At least it was before the onset of COVID-19 whose outbreak further fueled its growth when the use of digital services rose exponentially. This data deluge is unprecedented and has created new challenges for database management systems and file systems which are used in a wide range of applications for data analysis and management.

The traditional database management systems and file systems are developed considering the typical storage hierarchy where memory is fast but volatile and limited, and secondary storage is persistent and vast but has high latency. In such systems, the data is logically split into two sets of copies; working and consistent copies. The working copy resides in main memory, whereas the persistent copy resides on one or more secondary storage devices. Also, making data persistent is an error-prone process as problems like crashes and race conditions can corrupt data or leave it in an inconsistent state. There-

fore, strategies like journaling and copy-on-write are used to ensure the consistency of data. Moreover, the scalability of DRAM resulted in main memory database systems [2, 3, 4]. However, DRAM's further scalability has innately become quite a challenging task [5], and also, because of its energy consumption, the solution is unaffordable for most businesses.

Persistent memory, on the other hand, is considered to be an alternative to deal with the above-mentioned issues. It is a new category in the storage hierarchy that is non-volatile, byte-addressable, provides DRAM-like latency, and offers much higher capacity than DRAM. This new storage class has not only opened opportunities for new system designs but has also opened opportunities for enhancements in the existing storage engines. For instance, some work is already made in traditional database systems in that NVM is used to improve the traditional disk-based (centralized/decentralized) logging [6, 7, 8, 9]. In prior work, NVM is used as a buffer between DRAM and secondary storage devices [10, 5, 11]. Moreover, several index data structures like NVTree [12] and FPTree [13] are introduced that exploit the properties of NVM. Nevertheless, presently, there is no common storage engine that is built particularly to study the characteristics of the modern storage landscape, which has become more heterogeneous after the addition of NVM, and in consequence, there is a research gap in this direction.

In order to investigate the benefits of a common storage engine that manages all the storage devices in the modern storage landscape, a prototype of a general-purpose storage engine, called Haura [14, 15], is used.

It runs in user space and supports object and key-value interfaces. The key contributions of our work are as follows:

- We used PMDK (Persistent Memory Development Kit) to supplement Haura with persistent memory. PMDK is an open-source (C/C++) kit that offers different libraries/utilities to interact with persistent memory. We used the most appropriate library after a brief evaluation process.
- We investigated the impact of the above-mentioned change on Haura and used persistent memory to store the B^{ϵ}-tree nodes (Section 2.3).
- We identified the access patterns for different storage devices that supplement or affect the throughput of Haura.

The remainder of this paper is structured as follows. Section 2 provides background on non-volatile memory and the related programming techniques, and also briefly describes Haura. The implementation phase is discussed in Section 3 which is then followed by the evaluation in Section 4. Next, Section 5 details the related work and the paper concludes with a summary and open challenges in Section 6.

2. Background

In this section, we discuss non-volatile memory and different programming models to access it. We then describe Haura and briefly touch upon its key components.

2.1. Non-Volatile Memory

Persistent Memory (PMem), Storage Class Memory (SCM), Non-Volatile Memory (NVM), and Non-Volatile RAM (NVRAM) are the names often used to address this new class of storage. It sits between primary and secondary storage in the typical storage hierarchy, and it is also considered a disruptive technology as it has disrupted the traditional memory paradigm. It is non-volatile, has DRAM-like latency, and offers much higher capacity than DRAM. It is byte-addressable, and its property to be directly accessible using the cache lines by the CPU demands a different architecture than the one used in typical storage engines. Some example technologies are Phase Change Memory [16], Spin Transfer Torque RAM (STT-RAM) [17], Carbon NanoTube RAM (NRAM, NanoRAM) [18], and Memristors [19].

Presently, only Intel® produces persistent memory modules under the brand named Intel® Optane™ DC Persistent Memory. It offers different generations that vary in performance and capacity, and the modules are designed to be used with specific generations of Intel®'s processors, Intel® Xeon Scalable Processors, for instance.

They are available in DIMM form factor and compatible with conventional DDR4 sockets. They co-exist with conventional DDR4 DRAM DIMMs and use the same memory channel. The internal granularity of the modules is 256 bytes and they can be operated in three different modes; memory, app direct, and dual modes.

In the memory mode, NVRAM supplements DRAM where DRAM acts as an L4 cache and NVRAM as the main (volatile) memory. The host memory controller integrated into the processor manages the movement of data between DRAM and NVRAM. On the other hand, in the app direct mode, NVRAM is a persistent memory module where the applications have direct access to the device, it is still byte-addressable, and applications can use it as a storage device. Lastly, in the dual mode, part of the NVRAM can be allocated to applications, and the rest can be utilized as non-volatile memory.

2.2. Non-Volatile Memory Programming

The typical programming models categorize the data structures into two broad categories; memory resident and storage resident data structures [20]. It is mainly due to the underlying system architecture where main memory is attached directly to the memory bus and secondary storage, due to its high latency, communicates with the system via an I/O controller. The models operate on the data in main memory at byte granularity and ensure its persistency by explicitly writing it to secondary storage. A key challenge of such models is to ensure the consistency and integrity of data across all storage classes that share different characteristics. For example, the integrity of data in main memory could be ensured using mutexes whereas the consistency and durability of data on secondary storage are ensured using strategies like journaling and write-ahead logging [21].

The above-mentioned programming paradigm cannot be followed when working with persistent memory as it is attached to the memory bus and is non-volatile. Therefore, a new model is inevitable that simultaneously addresses all the atomicity and consistency issues, like concurrency and power failure, for instance [22, 20, 23]. Moreover, persistent memory is a comparatively new technology, and writing software with all the considerations requires an in-depth knowledge of the hardware and cache. Therefore, several APIs are available that handle the hardware-related intricacies internally. PMDK[1] (Persistent Memory Development Kit) is one such example which is based on SNIA NVM programming model[2].

---

[1] https://pmem.io/pmdk/
[2] https://snia.org/tech_activities/standards/curr_standards/npm

SNIA (Storage and Networking Industry Association) proposes different programming models [22, 24, 20, 23] to program persistent memory. The simplest way is to use the module as a block device and access it using a standard file API. Another approach is via an optimized file system, ext and XFS in Linux and NTFS in Windows, that is adapted specifically for persistent memory. This approach, contrary to the previous one, allows small read-and-write operations and is more efficient. Last but not least, DAX or Direct Access is another approach in that the persistent memory is accessed as a memory-mapped file. Nevertheless, contrary to memory mapping files on secondary storage, the operating system does not maintain pages for persistent memory in main memory.

2.3. Haura

Haura is a general-purpose and write-optimized tiered storage stack that runs in user space and supports object and key-value interfaces [14]. It has the ability to handle multiple datasets, provides data integrity, and supports advanced features like snapshots, data placement, and fail-over strategies. The core of the engine is the B^ϵ-tree, which is the sole index data structure in the engine. The engine supports block storage devices, solid-state drives, for instance, and also has its own caching layer using DRAM (separate from the operating system). It also offers features like data striping, mirroring, and parity. It follows ZFS [25] architecture and uses a similar layered approach [14] where layers or modules interact with each other using interfaces. A schematic diagram of Haura is presented in Figure 1.

Haura was initially built as a key-value storage engine where arbitrary-sized keys and values can be stored and an extension to support objects was made later in [15] in that the ObjectStore module was added to the stack. The ObjectStore module exposes the necessary routines to interact with the engine and supports all the primitive operations like create, read, write, and query. It uses the same key-value interface to store the objects. However, a key challenge it addresses is the transformation of objects into key-value pairs. Since, in the key-value version of Haura, although the keys and the values can have a variable size, there was still an upper limit defined on their sizes. Objects, on the other hand, can contain data of several gigabytes; therefore, a mechanism is devised in that objects are split into chunks, and each chunk is assigned a unique identifier. Moreover, the object name is primarily the key of the object, it can have a variable size and can spread over a few kilobytes, therefore, an indirection is added where the object name and other metadata are stored separately, and a unique fixed-size identifier is assigned to each chunk. Furthermore, the ObjectStore module, via the Database module, maintains two different datasets to store the data and the metadata
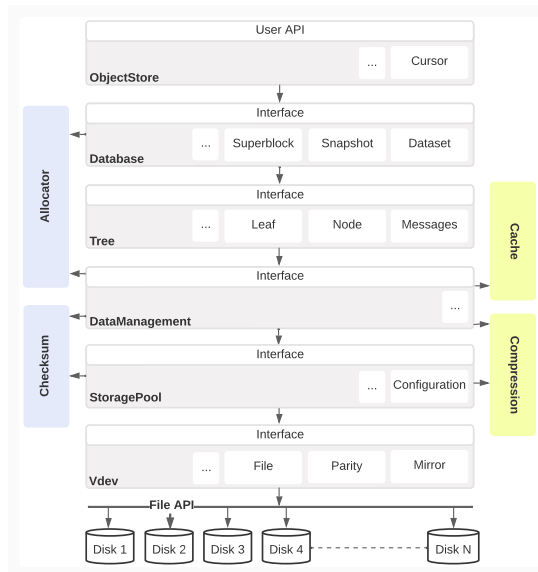
Figure 1: A layered conceptual diagram illustrating the main components of Haura. The objects in grey represent the main modules, whereas the objects in yellow and sky blue colors are the helper modules and classes respectively. The key classes in the modules are represented using white blocks.

of the objects. The first dataset stores the chunks of the object, whereas the second dataset stores the indirection-related information and other metadata, like modification time and size, to name a few.

The Database module controls and manages all the activities regarding a database. A database in Haura consists of one or more datasets and their respective snapshots. Datasets and snapshots are actually B^ϵ-trees. Moreover, it also maintains a separate B^ϵ-tree, named the root tree, to store all the information regarding the database. For example, it maintains active datasets and their pointers in the storage. It also maintains information regarding the usage of storage devices in the form of bitmaps.

The Tree module contains the actual implementation of the B^ϵ-tree and encapsulates all the tree-related operations and exposes the methods to the upper layer.

The DataManagement module ensures the persistence of the underlying data and its retrieval when requested. However, it internally interacts with a wide range of modules, especially with the helper modules, and plays a vital role in achieving their internal functionalities. The cache module, for instance, is managed by this module. The write and update requests from the upper layer first land into this module and then passed on to the cache module. Similarly, in the case of a cache miss, this module fetches the required data using the StoragePool module and are passes the data to the cache module for later

usage. Moreover, this module is also responsible for the compression and decompression of the data, and it uses the compression module for this purpose. Furthermore, the decision as to which blocks on storage media are to be used to write the data is also taken in this module, and it uses the AllocationHandler module to allocate the blocks. Last but not least, it communicates with the StoragePool module to perform the write and read I/O operations.

The StoragePool module performs two key operations. First, it maintains queues for asynchronous I/O operations. Second, it dispatches the I/O calls to the respective virtual devices in the Vdev module. However, it also exposes the methods for synchronous calls where it bypasses the queues. The interface of this layer matches with the Vdev module, however, it requires an additional parameter to communicate with the desired virtual device in the Vdev module as the module may contain more than one virtual device.

Lastly, the Vdev module provides different implementations to interact with the storage devices, and they are referred to as virtual devices in the system. Currently, Haura supports single, mirror, and parity implementations. The single version of the implementations works on a single storage device, and it has two further sub-implementations, file and memory, for SSD/HDD and DRAM (as volatile storage) respectively. It is the simplest implementation provided by this module, it is not fault-tolerant, and the underlying data is lost in case of error or failure. The other implementations, as their names suggest, mirror and parity, are introduced to support mirroring and parity functionalities respectively.

3. Implementation

In this section, we briefly discuss the implementation of the new virtual device for persistent memory and touch upon the important steps considered during this phase.

3.1. Programming Model Selection

PMDK provides several high and low-level libraries to interact with persistent memory. Haura, on the other hand, is also a well-developed engine and expects a virtual device to implement a certain interface. Therefore, in the initial phase, a list of properties is formulated to set a criterion, and each new implementation of a virtual device must adhere to the list to work properly with the existing interfaces in Haura. The properties are:

- Haura stores the nodes of the B^ϵ-tree using virtual devices, therefore, the virtual device should be able to perform read and write operations in varied block sizes, from a few kilobytes to megabytes.

- The virtual device should be able to perform both synchronous and asynchronous calls to the underlying storage device.
- Haura uses bitmaps to manage the allocation of the blocks on the storage devices. It partitions the whole space into equal-size blocks and allocates and de-allocates the blocks internally, therefore, this bookkeeping is not required in the virtual device or any library used in it.
- Haura uses the copy-on-write technique to update the nodes. It first copies nodes to the main memory, applies changes to the nodes, and then writes the data back to the device in a new location. It never performs in-place updates.

Now considering the above properties, libpmem and libpmemblk from PMDK's persistent library suite suits the current architecture of Haura and can be used to implement the functionality.

Libpmemblk is a high-level library that provides functionality to manage an array of fixed-size blocks. The blocks can be updated and read using their indices in the array. It does not provide byte-level access to the blocks, and any update requires the re-writing of the whole block. Whereas, libpmem is one of the low-level libraries in the kit, and the other high-level libraries are built on top of it. It wraps the basic operations exposed by an operating system and adds optimizations for persistent memory.

The other libraries from the PMDK's persistent suite, like libpmemobj and libpmemkv, are not a suitable choice because; first, they do not provide the required interface to implement the virtual device, and second, they internally perform the operations that are already addressed in Haura. For example, libpmemobj internally implements the object store functionality on top of memorymapped files, whereas the current architecture of Haura expects the virtual device to perform raw read and write operations on a specified location in the memory. Moreover, the management of key-value data at the library level, as in the case of libpmemkv, is the core functionality of Haura and is therefore redundant.

The final selection from the shortlisted[3] libraries is made using an experiment, which results are in Figure 2. First, it is quite evident from the graph that the approach for accessing persistent memory via a memory-aware file API is not a feasible approach as its latency to read and write the data, especially using small buffers, is significantly high. Nevertheless, a prominent drop can be observed in its latency with the increase in buffer size but it is still higher than the rest. The approaches that compete with each other are libpmem and libpmemblk. Libpmemblk, in some instances, performed better than libpmem, but its performance is worst with small buffers,

---

[3]PMem-aware file API is also added to the list for comparison.
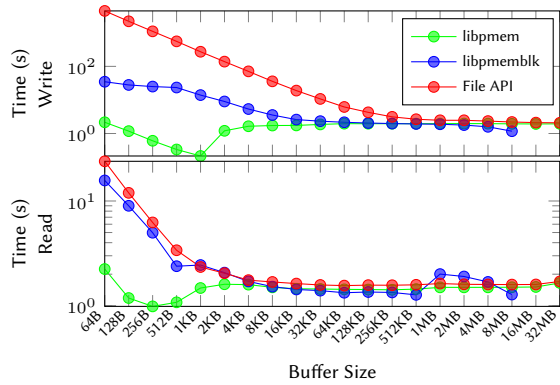
Figure 2: An object (size 5 GB) is written and read sequentially multiple times with different block sizes using libpmem, libpmemblk, and a standard File API.

and also it crashes when the block size exceeds 8 MB. Therefore, libpmem is finally selected as it is (comparatively) consistent throughout the experiment.

3.2. Rust Wrapper for libpmem

Haura is written in Rust, and PMDK supports other languages in that support for only C/C++ is fully tested, therefore, the second step, after selecting the library from PMDK's suite, involved writing a wrapper for the selected library (i.e. libpmem) in Rust. In this regard, Rust provides a utility named bindgen that generates the FFI[4] bindings to C/C++ libraries. The tool requires two files to generate the bindings. The first file is wrapper.h which should contain all the header files and declarations that the target application intends to use. The other file is build.rs which should contain the details regarding the generation of the bindings. This file is part of the folder structure followed in Rust and its compiler, before the compilation of the code, looks for this file in the root folder and executes it so that the bindings can be generated (and made available) before the execution of the actual program.

Once the bindings are generated, the next step involved writing the methods to perform read and write operations on persistent memory and exposing them to be used in its respective virtual device in the Vdev module which is mentioned in the following section.

3.3. NVM as a Virtual Device

As discussed in Section 2.3, Haura interacts with storage devices using different virtual device implementations in the Vdev module, and currently, there are four implementations available namely, file, memory, parity1, and

mirror. Similarly, a new implementation of a virtual device for persistent memory is added to the Vdev module. Moreover, as further mentioned in Section 3.1, the library from PMDK is chosen with careful consideration to avoid any architecture-related changes to Haura. Therefore, this new virtual device implementation exposes a similar interface as available in the other implementations and it internally makes use of the wrapper library mentioned in the previous section to perform the storage-specific operations.

The integration of the new virtual device required alterations in a few traits[5] and structs in different modules. For example, the DataManagement module interacts with the StoragePool module using a trait called `StoragePoolLayer`, and `StoragePoolUnit`, which is a struct in the StoragePool module, implements `StoragePoolLayer` and maintains the information regarding the configured virtual devices in an array of type `StorageTier`. Furthermore, the type `StorageTier` is an array of `Dev`, and `Dev` is an enum that stores the instance of the associated virtual device. The enum `Dev` provides three different types of features, `Leaf`, `Mirror`, and `Parity1`. `Leaf` further offers two different features, `File` and `Memory`. All these mentioned traits and structs are affected by the new implementation.

Furthermore, virtual devices are accessed using different traits that define distinct behaviors. The first trait is `Vdev`. It exposes functions to query different properties and states of the virtual devices. For example, it can be used to fetch the id and size of the device. On the other hand, the traits `VdevWrite` and `VdevRead`, as their name suggests, are used to perform read and write operations on virtual devices, and provide methods to perform the operations synchronously and asynchronously. These traits are implemented for the new virtual device. Last but not least, other changes have been added to make the virtual device visible to Haura through configuration details.

4. Performance Analysis

In this section, we analyze the impact of the newly added virtual device on Haura. We start by testing Haura for different workloads, and the baseline is the existing best-performing implementation of the virtual device. We then study the impact with different configurations, with different thread counts and cache sizes, for instance.

4.1. Experimental Setup

The experiments are performed on a dual-socket server having Intel® Xeon® Gold 5220R with 2.20 GHz base fre-

---

[4]Foreign function interface (FFI) is a method to invoke calls from a library written and compiled in a different language.

[5]A trait, in Rust, can be considered as an equivalent to an interface in object-oriented languages like Java and C# [SK22d].
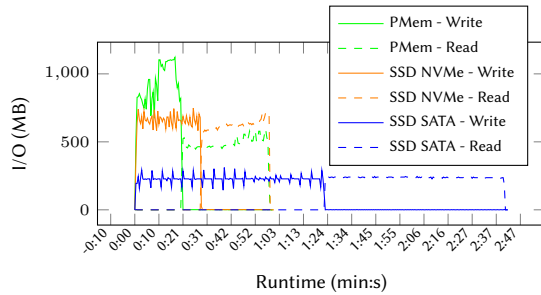
Figure 3: Three different executions representing the sequential I/O throughput of Haura when configured with different storage devices. The data is recorded at an interval of 500ms.

quency and each CPU contains 24 physical cores and each core supports two threads. Each CPU-socket contains two integrated memory controllers (iMCs) with three memory channels, each channel (except the last ones) connected to one PMem and DRAM DIMM resulting in 4 interleaved[6] PMem and 6 DRAM DIMMs per socket. The PMem used is 128 GB Intel® ™ DC Persistent Memory Series 100 DIMMs, resulting in a total persistent memory capacity of 1 TB (128 GB x 4 DIMMs x 2 sockets), and the capacity of DRAM is 384 GB (32 GB x 6 DIMMs x 2 sockets). Moreover, the server contains two NUMA nodes each with 48 logical cores, 4 PMem DIMMs, and 6 DRAM DIMMs. However, to avoid memory access overhead, the experiments are run on socket 0. Lastly, the machine runs Ubuntu 20.04.3 LTS (5.4.0-126-generic), and PMem is accessed in the app direct Mode using fsdax [7].

4.2. Sequential Workload

In this experiment, Haura is configured for three different storage devices; PMem, SSD NVMe, and SSD SATA. The experiment writes 5 objects, each size 5 GB, and then reads them sequentially in the same order, however, the write requests are asynchronous, which allows multiple write requests to be dispatched, whereas the read requests are synchronous.

The results in Figure 3 show that PMem performed better than the rest for the write I/O, and it lagged behind SSD NVMe for the read I/O. But, the resulted throughput for PMem in both cases is quite off from the expected values because as per the specifications, a single PMem DIMM (with four cache-lines) can write and read up to 1,800 and 6,800 MB/s[8], respectively[9].

The key reason behind the poor performance of the read I/O is the layout in which Haura stores the data. Haura starts by splitting the objects into chunks and transforming them into messages. The messages are then pushed into the root node, and they descend gradually to the target leaf node, and during the descent, they are buffered in internal nodes and flushed down to the child node only when the buffer is full. Later, when the sync operation is performed, Haura follows the postorder [26] approach to persist the data that does not guarantee the ordering of the chunks on the storage device. On the other hand, when Haura fetches an object, it starts fetching its chunks sequentially from the root node first and keeps fetching the child nodes until it reaches the leaf node or finds the messages for the queried chunk. Therefore, this reading approach cannot benefit from sequential access as the tree data is already stored in the postorder layout. Furthermore, the other main reason that applies to both scenarios is the use of a single thread that leaves the device underutilized. Last but not least, the reason the write I/Os performed better is because they were asynchronous calls where the thread was capable of issuing multiple asynchronous I/Os using the asynchronous programming technique, whereas the read I/Os were synchronous calls.

Moreover, another interesting pattern that surfaced during the detailed analysis is, the relative performance of the storage devices was not consistent all the time. As shown in Figure 4, the difference between PMem and SSD NVMe is significant for small block sizes, however, the difference shrinks considerably for large blocks.

4.3. Random I/O and Worker Threads

This experiment evaluates the impact of the number of threads and cache size on Haura when configured with different storage devices. It writes an archive file[10] (size 1011 MB) as an object to the engine. The file contains 80,690 entries with metadata stored in the first 9.3 MiB that contain the central directory to locate the individual files. Moreover, the scenarios with circles (Figure 5) store the metadata on the first device (i.e. SSD SATA) and the remaining content on the second mentioned device. Lastly, the script fetches 50,000 files randomly[11].
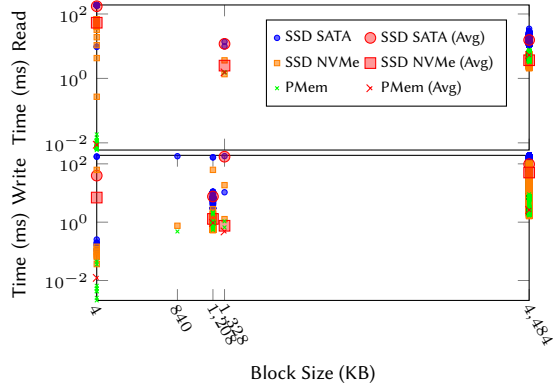
The results are presented in Figure 5, and it is evident from all sub-plots that the execution time improves with the increase in the worker threads and cache size.

The scenarios that performed worst are the ones that used SSD SATA to store the contents and a faster de-

---

[6]In DIMM interleaving, the data is interleaved as per the configured block size (i.e., 4 KB in the current settings.) across the DIMMs.
[7]https://docs.pmem.io/ndctl-user-guide/managing-namespaces
[8]https://www.intel.de/content/www/de/de/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html
[9]SSD NVMe can perform sequential read and write operations of up

to 3,200 and 2,000 MB/s and random read and write operations of up to 540,000 and 55,000 IOPS. SSD SATA can perform sequential read and write operations of up to 550 and 510 MB/s and random read and write operations of up to 86,000 and 30,000 IOPS, respectively.
[10]https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.12.13.tar.xz
[11]https://docs.rs/xoshiro/latest/xoshiro/struct.Xoshiro256Plus.html

Figure 4: The plots group the calls (write and read respectively) from Figure 3 with respect to their payloads.
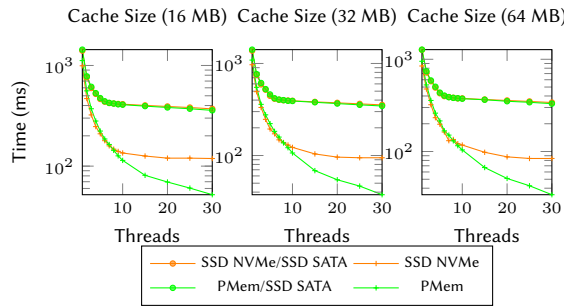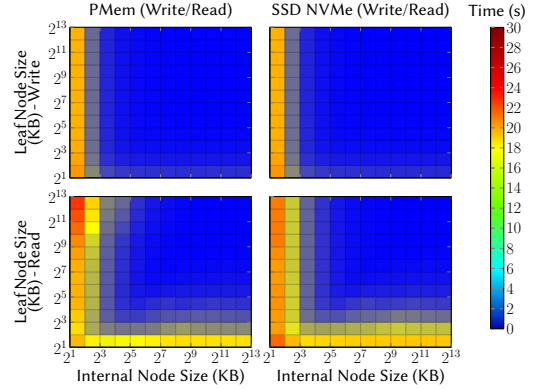

Figure 5: The impact of threads and different cache sizes on Huara when used with different storage configurations.

vice to store the metadata of the file. However, a minute difference can be observed, with PMem the execution time is slightly worse for threads fewer than 6, nevertheless, the difference diminishes, and with the increase in threads (e.g., 30 threads), PMem performed better than SSD NVMe. On the other hand, a significant difference in performance can be seen when only PMem and SSD NVMe are used to store the whole file. As can be seen, SSD NVMe performs marginally better when fewer threads are used, however, as the thread count passes 9 threads, PMem surpasses SSD NVMe with a significant difference at the end.

4.4. Node-Size Significance

An intriguing behavior we came across while discussing the sequential workload is that the size of the payload influences the performance of Haura. Therefore, to further investigate the behavior, this experiment evaluates Haura (for PMem and SSD NVMe) with different internal and leaf node sizes, that is, thus far set to 4 MB for both node types. The experiment first sets the size of the inter-

nal nodes and then repeats the experiment with different leaf node sizes, and writes and reads an object with the size of 128 MB sequentially.

The results are illustrated using a heatmap in Figure 6. First, it is evident that both storage devices share almost the same temperatures. Second, the engine performs worst for small block sizes, especially for internal nodes. However, the performance improves with the increase in the size, and the concentration of blue color indicates the engine performs better under large block sizes.

One reason for the high temperatures is due to the height of the tree that grows deep when the nodes, internal nodes in particular, are small. For instance, when the node size is 512 bytes, an object size 128 MB would result in 26,1376 nodes[12], and the internal nodes contain a limited number of messages and pivots, whereas, when the node size is 4 MB, the tree would only need 32 nodes to accommodate the object. Therefore, when the tree is deep, Haura spends considerable time flushing and merging the nodes. Moreover, another obvious reason is when the nodes are small multiple requests are dispatched to the storage devices. However, further analysis is required to capture the time only taken by the virtual devices.

5. Related Work

In existing engines, NVM is mostly utilized to improve the caching and recovery of the engines. In [6, 7], the logging component uses NVM at different levels that improve the logging and recovery of the engine. Moreover, [27] discusses three different logging techniques and implements their equivalent NVM designs. The results show that in-place update is the most appropriate technique for NVM. Furthermore, SOFORT [28] and FOE-

Figure 6: Multiple end-to-end executions to analyze the impact of internal and leaf node sizes on the throughput of Haura.

---

[12]The actual count of the nodes for an object size 128 MB is higher than 261376 because each object chunk is assigned a key as well.

DUS [29] are examples of main memory database systems that utilize NVM to improve the recovery of the system.

In some literature, NVM is also utilized as a buffer and there are two main designs in this approach. The first is to use NVM to supplement DRAM which is already mentioned in [28, 29, 30], and the second is to use NVM as another layer between DRAM and secondary storage and in this regard, a technique called three-tier buffer management is suggested in [31].

Furthermore, data structures are also optimized to exploit the full potential of NVM. FPTree [13] is an NVM-aware B$^+$-tree that stores leaf nodes in NVM and internal nodes in DRAM, and it performs better than other NVM-optimized trees, NV-Tree [12] and wBTree [32], for instance. Moreover, FOEDUS [29] also uses a customized tree called Master-Tree.

Our work enables Haura to persist the tree nodes on NVM, which, along with an allocation strategy, can be used to improve recovery and caching. However, the migration of the persisted nodes is presently not possible. Haura can also store the internal nodes on NVM as done in FPTree [13]. Lastly, depending on the size of the data, the engine can be used as NVM-DRAM engine.

6. Conclusion

In this work, Haura, a general-purpose and write-optimized storage engine is used to study the characteristics of the modern storage landscape that has become more heterogeneous with the advent of PMem which is a promising technology that shares the characteristics of primary and secondary storage and has disrupted the traditional memory paradigm. A few important findings our work uncovered are; first, persistent memory performs optimally when accessed using the largest possible blocks in random workloads. Second, the size of the cache and thread count impact Haura's throughput. Last but not least, the size of the nodes also determines the throughput of the engine with the internal node having more influence. The insights gathered in this paper can be used to significantly improve Haura's performance and further exploit the characteristics of PMem. However, two aspects that need to be investigated are the use of in-place updates for PMem and accessing it using devdax[13] that produces better results than DAX in certain cases [33].

References

[1] U. Kazemi, A survey of big data: challenges and specifications, CiiT IJSETA (2018).

[2] F. Faerber, et al., Main memory database systems, Foundations and Trends® in Databases (2017).

[3] P.-Å. Larson, J. Levandoski, Modern main-memory database systems, VLDB Endowment (2016).

[4] J. DeBrabant, et al., Anti-caching: A new approach to database management system architecture, VLDB Endowment (2013).

[5] I. Oukid, et al., Storage class memory and databases: Opportunities and challenges, it-Information Technology (2017).

[6] J. N. Gray, Notes on data base operating systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 1978, pp. 393–481.

[7] T. Wang, et al., Scalable logging through emerging non-volatile memory, Proceedings of the VLDB Endowment 7 (2014).

[8] D. Lomet, R. Anderson, et al., How the Rdb/VMS data sharing system became fast, DEC Cambridge Research Lab Technical Report CRL 92 (1992).

[9] J. Huang, K. Schwan, M. K. Qureshi, NVRAM-aware logging in transaction systems, VLDB Endowment (2014).

[10] P. Götze, et al., Data management on non-volatile memory: a perspective, Datenbank-Spektrum (2018).

[11] A. Eisenman, et al., Reducing DRAM footprint with NVM in Facebook, EuroSys, 2018.

[12] J. Yang, et al., NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems, USENIX FAST, 2015.

[13] I. Oukid, et al., FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for SCM, MOD, SIGMOD, 2016.

[14] F. Wiedemann, Modern Storage Stack with Key-Value Store Interface and Snapshots Based on CoW B$^\epsilon$-Trees, 2018.

[15] T. Höppner, Design and Implementation of an Object Store with Tiered Storage, 2021.

[16] A. Faraclas, et al., Modeling of set and reset operations of phase-change memory cells, IEEE electron device letters (2011).

[17] A. K. Mishra, et al., Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs, ISCA, IEEE, 2011.

[18] B. Gervasi, Will carbon nanotube memory replace DRAM?, IEEE Micro (2019).

[19] D. B. Strukov, G. S. Snider, D. R. Stewart, R. S. Williams, The missing memristor found, nature (2008).

[20] S. Scargall, Programming persistent memory: A comprehensive guide for developers, Springer Nature, 2020.

[21] S. Sippu, et al., Transaction processing: Management of the logical database and its underlying structure, Springer, 2015.

[22] A. Baldassin, et al., Persistent Memory: A Survey of Programming Support and Implementations, ACM CSUR (2021).

[23] A. Rudoff, Persistent memory programming, Login: The Usenix Magazine (2017).

[24] A. Rudoff, Programming models for emerging non-volatile memory technologies, USENIX & SAGE (2013).

[25] W. Fuzong, G. Helin, Z. Jian, Dynamic data compression algorithm selection for big data processing on local file system, in: Proceedings of the International Conference on Computer Science and Artificial Intelligence, 2018, pp. 110–114.

[26] G. Valiente, Algorithms on trees and graphs, volume 112, Springer, 2002.

[27] J. Arulraj, et al., Let's talk about storage & recovery methods for non-volatile memory database systems, SIGMOD, 2015.

[28] I. Oukid, et al., SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery, DaMoN, 2014.

[29] H. Kimura, FOEDUS: OLTP engine for a thousand cores and NVRAM, ACM MOD, SIGMOD, 2015.

[30] L. Lersch, et al., An analysis of LSM caching in NVRAM, DaMoN, 2017.

[31] A. van Renen, et al., Managing non-volatile memory in database systems, MOD, 2018.

[32] S. Chen, et al., Rethinking database algorithms for phase change memory, Cidr, 2011.

[33] B. Daase, et al., Maximizing persistent memory bandwidth utilization for OLAP workloads, PODS SIGMOD, 2021.

---

[13]https://docs.pmem.io/ndctl-user-guide/managing-namespaces