Storage System Designs with Emerging Storage Technologies

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Baoquan Zhang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Prof. David H.C. Du

January, 2021

Dedication

"Perhaps this life was not my true life, this world was not my true home. But she was."
–Akecheta, WestWorld

Abstract

In modern data centers, the volume of data has grown to an enormous size with an incredible speed due to the flourish of the Internet, mobile network, and the Internet of Things (IoT). Storage systems play a critical role under different scenarios, e.g., machine learning pipeline, interactive data analysis, data storage service, etc. Applications have to meet with very high requirements in the aspects of performance, capacity, and reliability. However, the I/O performance of storage systems suffer from the long-latency of storage devices. Besides, the data area density of storage devices has reached a bottleneck. Thus, it becomes difficult to increase the capacity of storage systems further. At last, silent data corruption happens more frequently than we expect. Traditional methods, e.g., replica, erasure code, etc., are not sufficient to ensure data reliability anymore.

To address these challenges of performance, capacity, and data reliability in storage systems, storage vendors have proposed new storage technologies/devices. Firstly, Non-Volatile Memory (NVM) is a persistent memory that provides memory-speed data persistence and byte-addressable data accesses. Secondly, Shingled Magnetic Recording (SMR) is a promising layout to increase data area density further with existing magnetic recording technologies. At last, T10 Protection Information (T10-PI) drives are proposed against data corruption. However, current storage systems need to be optimized or even redesigned to leverage the advantages of these new storage technologies/devices.

This thesis introduces four complemented research topics targeting designing new storage systems using NVM, SMR drives, PI-capable drives, and hybrid systems with NVM and storage devices, respectively. NVLSM is a key-value store using Log-Structured Merge Tree (LSM-Tree) on NVM systems. Idler is a mechanism to control the I/O workload to minimize the tail response time of an SMR drive. Idler artificially induces idle cleanings to avoid expensive blocking cleanings. DIX-aware RAID improves the data integrity in Linux software RAID using T10-PI against any data corruption during data transmission and persistence. Finally, PMDB is a new key-value store on systems with both NVM and traditional storage devices to achieve performance and capacity simultaneously.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the flourish of Internet, mobile network, and the Internet of Things (IoT), data is generated from many sources at an incredible rate. In modern data centers, applications rely on storage systems to preserve, store and retrieve these generated data. Storage systems have played a critical role in many scenarios, e.g., data storage services [1], cloud computing [2], machine learning pipeline [3], real-time data analysis [4], etc. These applications have introduced new challenges to the modern storage systems in the aspects of *performance*, *capacity*, and *reliability*.

Performance More and more upper-layer applications require storage systems to provide low-latency and high-throughput data accesses. For instances, worldwide people are running billions of analyses over Google's 100 petabyte index every day [5]. In Facebook, 136,000 photos are uploaded, 510,000 comments are posted, and 293,000 status updates are posted every 60 seconds in United States only [6]. Those operations typically require the responses of the back-end storage systems under millisecond-level. The current storage systems suffer the long I/O latencies from the persistent storage devices. A widely-used solution is to utilize DRAM buffers to achieve fast responses [7]. However, the high-performance DRAM is not persistent and will lose all data after a power failure or system crash.

Capacity Applications are continuously generating more data. The most known example concerning Google's database mentions the total gross estimate of all data Google saved by 2016 as approximately 10 EB. A. Orlova stated in 2015 that Facebook generates about 10 TB every day, and Twitter generates about 7 TB. Some enterprises

generate TB every single hour. In general, the digital universe is doubling in size every two years, and by 2020, the data created and copied annually will reach 44 ZB or 44 trillion GB [8]. However, the data area density of conventional disk drives has reached a bottleneck. It becomes hard to increase the capacity of the storage device further. Therefore, storage systems are facing new challenges to handle the massive data generated every day.

Reliability Storage systems are required to provide reliable data storage service. Therefore, mechanisms, e.g., data replica [9], erasure code [10], etc., are widely used in large-scale storage systems. However, silent data corruptions [11] happen more frequently than we expect in a large scale storage system due to the frequent data transmission and migration [12]. Storage systems may have data being corrupted without any error messages. Therefore, the traditional methods, e.g., data replica, erasure code, etc., are not sufficient to ensure data reliability anymore.

To address the above challenges, storage vendors propose new storage technologies/devices. First of all, Non-Volatile Memory (NVM) [13] improves the I/O performance of storage systems significantly. With NVM, storage systems can provide memory-speed data persistence. Secondly, Shingled Magnetic Recording (SMR) [14] technology increases data area density further with the existing Perpendicular Magnetic Recording. The capacity of a storage system has been enlarged greatly by using SMR layout. At last, Technical Committee T10 [15] proposes Protection Information (PI) [16] to against silent data corruption. PI-capable drives can guarantee end-to-end data integrity in storage systems.

However, the existing storage systems need to be optimized or redesigned to utilize the new features of these emerging storage technologies. Therefore, this thesis introduces four completed research topics to design storage systems utilizing the three, as mentioned above, storage technologies, respectively. NVLSM is a key-value store using Log-Structured Merge Tree (LSM-Tree) on NVM systems. NVLSM utilizes a type of accumulative compaction and a fractional cascading searching by fully leveraging the byte-addressable data accesses of NVM. Therefore, the write amplification can be reduced without significantly increasing the read amplification. Idler is a mechanism to control the I/O workload to minimize the tail response time of an SMR drive. Idler artificially induces idle cleanings to avoid the expensive blocking cleanings. DIX-aware

RAID improves the data integrity in Linux software RAID to allow PI to be verified, passed in the kernel RAID module, and stored together with data on PI-capable drives. Any data corruption, including silent data corruption, can be detected during the processes of data transmission and persistence. Finally, PMDB is a key-value store built on systems with both NVM and traditional storage devices to achieve performance and capacity simultaneously.

The rest parts of the proposal are organized as follows:

- Chapter 3 discusses NVLSM, a persistent memory key-value store with LSM-Tree using accumulative compaction.
- Chapter 4 presents Idler, a I/O workload controlling mechanism for better responsiveness on SMR drives.
- Chapter 5 talks about how to improve data integrity in Linux Software RAID using Protection Information.
- Chapter 6 demonstrates a efficient range-based key-value store on hybrid systems with both NVM and traditional storage devices.
- Chapter 7 offers conclusions.

Chapter 2

Emerging Storage Technologies

2.1 Non-Volatile Memory (NVM)

The emerging NVM technologies, like Spin-Transfer Torque Memory (STT-RAM) [17], Phase Change Memory (PCM) [18], 3D XPoint [19], etc., are changing the architecture of storage systems. The access latency of STT-RAM can be similar or even smaller than that of DRAM. STT-RAM may appear in on-chip or last level-cache since its capacity is limited by the large cell size [20]. Both PCM and 3D XPoint can achieve a higher density than that of DRAM while their access latency is several times larger than that of DRAM [21]. Therefore, current NVM technologies cannot totally replace DRAM. In this paper, we target persistent memory systems with a hybrid main memory architecture including a large but a bit slower NVM sitting on the memory bus together with a small and fast DRAM. Comparing with the systems using traditional storage devices, e.g., hard disks or flash drives, persistent memory systems provide a higher performance with low-latency and byte-addressable data accesses.

Building systems for NVM is challenging since NVM suffers from an asymmetric performance for read and write. An unexpected shutdown may result in data inconsistency due to the small atomic update granularity and the instruction reorders [22, 23, 24, 13, 25]. Writes to NVM is typically protected by some mechanisms, e.g. write-ahead log, copy-on-write, log-structured updates combining with *mfence* [26] and *clflush* [27] instructions [28, 29, 30, 31, 32, 33]. Besides, the write latency of NVM can be multiple times larger than that of reads. The performance overhead of ensuring data

consistency during updates should be minimized to fully utilize the high performance of NVM systems. Besides, certain types of PCM-based devices have shorter endurance. With a large number of write operations, the lifetime of an NVM devices can be significantly shortened [34, 35]. Therefore, applications running on NVM systems have to minimize their write operations to NVM.

2.2 Host-Aware SMR Drives

The internal structure of Host-Aware SMR Drives is shown in Figure 2.1. It includes STL and SMR zones. An SMR zone is a collection of overlapped data tracks [36]. STL consists of a media cache and a mapping table. The buffered data in the media cache are log-structured since the media cache also consists of shingled tracks. Each shingled zone has a writing position called a write pointer. Any write beginning with the position pointed by the write pointer is considered as a *Sequential SMR Write* and can be issued directly to its designated location. The position of the write pointer will also be updated accordingly after a sequential write. A write with its targeted address not beginning at the write pointer is called a *Non-Sequential SMR Write*. A non-sequential write will be re-directed to media cache by STL. Its designated zone will be changed from a sequential state to a non-sequential state. The write pointer is no longer valid in a non-sequential zone. All the following write requests to a non-sequential zone will be buffered in the media cache [37].

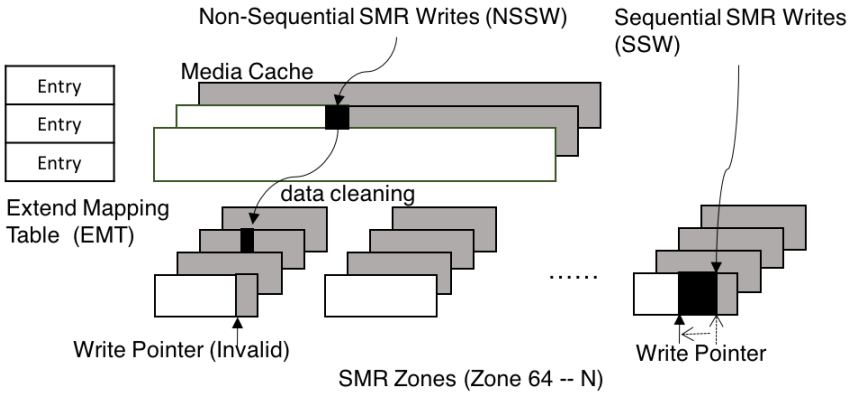


Figure 2.1: Internal Structure of HA-SMR Drives

2.3 T10 Protection Inforamtion

T10 PI enables applications and operating systems to generate Protection Information (PI) along with I/O requests, and verifying it at each layer of the I/O stack from the source of the request to the disk. Figure 2.3 demonstrates the key layers in I/O stack protected by the DIF and DIX. Since operating systems deal with data in page sizes, the scatter-gather list structure is utilized to store the data and PI in memory and later merged in the HBA [38].

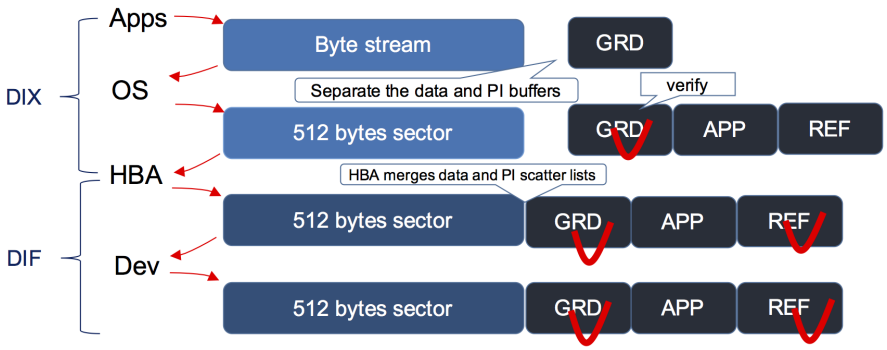


Figure 2.2: I/O stack protected by DIF/DIX

Chapter 3

NVLSM: A Persistent Key-Value Store using Log-Structured Merge Tree using Accumulative Compaction

3.1 Introduction

The emerging Non-Volatile Memory (NVM) technologies such as 3D XPoint [19], Spin-Transfer Torque Memory (STT-RAM) [17] etc., promise data persistence in memory access speed. An NVM device can be used as both memory and storage, thus leads to a persistent memory system. Although NVM is byte-addressable, the current NVM cannot totally replace DRAM due to its relatively large read/write latency. Therefore, a small but fast DRAM may still be kept in an NVM system for peripheral data [39, 40, 41, 42].

Even though NVM provides much higher performance comparing to that of magnetic disk or flash memory based drives, building systems using NVM is challenging since data consistency can be a problem when power failed or system crashed. The introduced performance overhead to guarantee data consistency for writes may counteract the performance benefits of NVM [43, 44, 45, 46]. Besides, certain types of NVM

devices suffer from short endurance. Applications running on NVM devices have to minimize the number of write operations to increase the lifetime of NVM devices [35].

In a modern storage infrastructure, key-value stores, like LevelDB [47] and RocksDB [48] using Log-Structured Merge Tree (LSM-Tree) [49], have attracted more attentions for write-intensive workloads [50, 51, 52]. Existing studies have used NVM to improve the performance of LSM-Tree based key-value stores [53, 54, 55]. Therefore, deploying LSM-Tree on persistent memory systems to achieve great performance improvements is critical to many upper-layer applications.

An LSM-Tree organizes key-value pairs into multiple components with increasing data capacity of each component. Note that we use the term "component" to distinguish the "level" used in LevelDB. New key-value pairs will first be inserted into the smallest component. A process called compaction migrates key-value pairs to next component if the size of the current component has reached to a threshold. However, the most widely-used leveled compaction migrates data with read-merge-writes leading to a large write amplification. That is, the size of data written to the NVM device is multiple times larger than that from applications. In persistent memory systems, a large write amplification considerably degrades the insert performance due to the data consistency overhead and potentially shortens the lifetime of NVM devices. Therefore, it is crucial to reduce the write amplification if an LSM-Tree based key-value store intends to fully benefit from using NVM.

Some other LSM Tree-based key-value stores use a tiered compaction to achieve a small write amplification, but they significantly increase read amplification. Besides, tiered compaction requires a larger transient space for a single compaction [56, 57]. Monkey [58] and Dostoevsky [59] have discussed the trade-offs among overheads of write, read and space. PebblesDB [60] employs a fragmented compaction to reduce write amplification. Fragmented compaction can achieve a similar write amplification as tiered compaction while limiting the required transient space for a single compaction. However, it still results in a relatively large read amplification. In this paper, we intend to answer the following question: *Can we reduce the write amplification in an LSM Tree based key-value stores using NVM without significantly increasing read amplification with similar space overhead as in fragmented compaction?*

To address this question, we propose **NVLSM**: a design of LSM-Tree for persistent

memory systems using a new compaction scheme called Accumulative Compaction. Since NVM provides more flexibility with its byte-addressable data accesses, Accumulate Compaction fully leverages the byte-addressable data accesses of NVM by including a pointer with each key-value pair. In Accumulative Compaction, a sorted run of key-value pairs is migrated from one component to the next larger component by building a new floor using these pointers instead of using read-merge-writes to re-write data. A real compaction will only be carried out when the number of floors reach to a threshold. This thereby achieves a small write amplification with similar space amplification as fragmented compaction. Besides, NVLSM constructs fine-grained links among key-value pairs between floors to enable a cascading searching to limit/reduce read amplification.

With Accumulative Compaction, NVLSM achieves small write and read amplifications simultaneously. The threshold of the maximum number of floors can be used to obtain a trade-off between write and read amplifications. That is, the larger the threshold, the write amplification is reduced, however, the read amplification is increased. We compare NVLSM with key-value stores using LSM-Tree with leveled compaction and fragmented compaction. The results indicate NVLSM is able to reduce the write amplification by up to 67% compared with LSM-Trees using leveled compaction. It achieves a small write amplification close to that of LSM-Trees using fragmented compaction and a smaller read amplification comparable with that of LSM-Trees using leveled compaction. In a read-modify-write workload, NVLSM outperforms other key-value stores by 1.92× – 2.56×.

3.2 Background

3.2.1 Persistent Memory Systems

The emerging NVM technologies, like Spin-Transfer Torque Memory (STT-RAM) [17], Phase Change Memory (PCM) [18], 3D XPoint [19], etc., are changing the architecture of storage systems. The access latency of STT-RAM can be similar or even smaller than that of DRAM. STT-RAM may appear in on-chip or last level-cache since its capacity is limited by the large cell size [20]. Both PCM and 3D XPoint can achieve a higher density than that of DRAM while their access latency is several times larger than that of DRAM [21]. Therefore, current NVM technologies cannot totally replace

DRAM. In this paper, we target persistent memory systems with a hybrid main memory architecture including a large but a bit slower NVM sitting on the memory bus together with a small and fast DRAM. Comparing with the systems using traditional storage devices, e.g., hard disks or flash drives, persistent memory systems provide a higher performance with low-latency and byte-addressable data accesses.

Building systems for NVM is challenging since NVM suffers from an asymmetric performance for read and write. An unexpected shutdown may result in data inconsistency due to the small atomic update granularity and the instruction reorders [22, 23, 24, 13, 25]. Writes to NVM is typically protected by some mechanisms, e.g. write-ahead log, copy-on-write, log-structured updates combining with *mfence* [26] and *clflush* [27] instructions [28, 29, 30, 31, 32, 33]. Besides, the write latency of NVM can be multiple times larger than that of reads. The performance overhead of ensuring data consistency during updates should be minimized to fully utilize the high performance of NVM systems. Besides, certain types of PCM-based devices have shorter endurance. With a large number of write operations, the lifetime of an NVM devices can be significantly shortened [34, 35]. Therefore, applications running on NVM systems have to minimize their write operations to NVM.

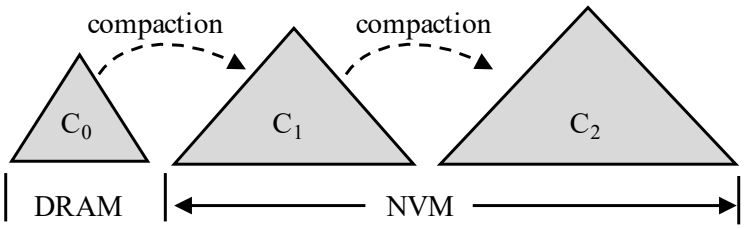


Figure 3.1: LSM Tree

3.2.2 LSM-Tree on Persistent Memory Systems

LSM-Tree is a write-optimized data structure widely used in many key-value stores [50, 51, 61, 47]. Figure 3.1 shows one possible structure of LSM-Tree on persistent memory systems. LSM-Tree organizes key-value pairs into multiple components (or levels) with increasing sizes by a configurable size ratio between two adjacent components. New key-value pairs are only inserted to the smallest component C_0 which is

typically small enough to be kept in DRAM. Then a data migration process called compaction migrates data from a smaller component to next larger component when the size of the smaller component reaches to a threshold. Existing studies have used NVM to improve the performance of LSM-Tree-based key-value stores [53, 62]. Further deploying LSM-Tree-based key-value stores on persistent memory systems to achieve great performance improvements is critical to many upper-layer applications.

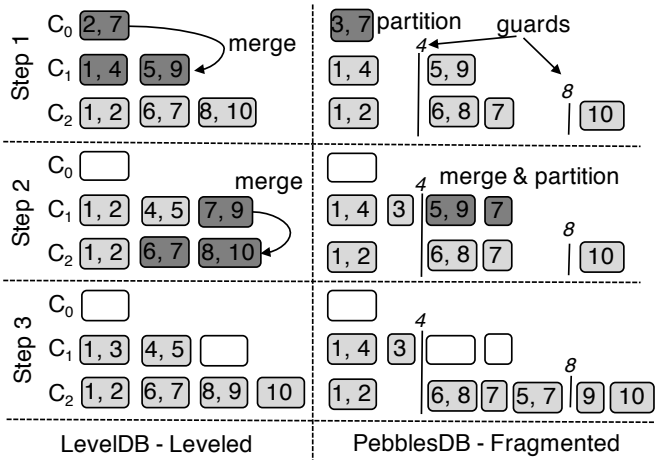


Figure 3.2: LSM-Tree compaction strategies.

However, the most widely-used leveled compaction in LevelDB [47] leads to a large write amplification, i.e. the data size written to NVM is multiple times larger than that from user applications. The left side of Figure 3.2 shows an example to demonstrate the leveled compaction process. In LSM-Tree with leveled compaction, a component includes multiple sorted runs with disjoint key ranges. Each sorted run consists of multiple key-value pairs in a sorted order with a bounded total size. There is a fixed size ratio between the numbers of the sorted runs in two adjacent components. In our example, for the purpose of explanation, the size ratio is 2. C_0 is the smallest component in DRAM. Its size is set for a sorted run. When migrating data, the leveled compaction is executed with a rolling-merge process to maintain the disjoint key ranges.

The read amplification for an LSM-Tree using leveled compaction is small since only one sorted run needs to be checked for each component during a search. While, the rolling merge operation leads to a large amount of data rewrites on the same component.

Monkey [58] and Dostoevsky [59] have analyzed the operations of leveled compaction and found that a key-value pair may be rewritten multiple times on the same component. The large amount of data rewrites seriously increases write amplification. Considering the high write overhead in persistent memory systems as we have discussed, a larger write amplification degrades the insertion/update performance significantly.

Tiered compaction is able to migrate data between components with a smaller write amplification but it results in a larger read amplification for searching. Tiered compaction is less used since it also requires a larger transient space for compaction operation [57, 56]. A type of fragmented compaction (the right part of Figure 3.2) is proposed by PebblesDB [60] to reduce write amplification. In fragmented compaction, components are partitioned with guard keys. A guard in a smaller component will also be a guard for larger components. The number of guards in a component increases with the growing of the component size. Sorted runs in the same guard can have overlapped key ranges.

Besides the size ratio of components, PebblesDB also sets a maximal number of sorted runs in a guard. When reaching to the maximal number, sorted runs in a guard will have to be merged and sorted. If the current component is the largest component and its size is still smaller than the predefined capacity, the merge results will be rewritten to the current component. Otherwise, the merge results will be partitioned based on the guards of the next component and written to the next component (i.e., a new component is created).

Fragmented compaction reduces the write amplification significantly since data will not be rewritten in the same component. Besides, it requires a smaller transient space and time for a single compaction comparing with tiered compaction. However, fragmented compaction still introduces a relatively larger read amplification. The performance of searching operations is negatively impacted since multiple sorted runs have to be potentially checked in each component. Since fragmented compaction performs better than tiered compaction, in this paper we will mainly compare the proposed Accumulative Compaction with fragmented compaction.

Monkey in [58] and Dostoevsky in [59] have analyzed the overheads of leveled and

tiered compaction on storage systems. Persistent memory systems have different properties due to its capability of byte-addressable data accesses. We analyze the amplifications of read, write and space for both leveled compaction and fragmented compaction for persistent memory systems. In our analysis, we consider the worst case of write, read and space amplifications.

In an LSM-Tree containing $\boldsymbol{N}$ number of key-value pairs and $\boldsymbol{F}$ number of key-value pairs in the first component. Monkey [58] has identified that there will be $\log_T(\frac{N}{F} \cdot \frac{T-1}{T})$ components if the size ratio of adjacent components is $\boldsymbol{T}$. In addition, we assume the size of one key-value pair is $Size_{kv}$ and the maximal number of key-value pairs in one sorted run is $\boldsymbol{S}$. For fragmented compaction, we have at most $\boldsymbol{M}$ overlapped sorted runs in a guard.

Write Amplification We define the Write Amplification (WA) as the ratio of data size written to NVM to that from applications. In leveled compaction, one key-value pair may be rewritten in the same component up to T times [58, 59]. Therefore, the size of NVM writes for one key-value pair is $T \cdot Size_{kv}$ in each component. The total WA is $\frac{T \cdot Size_{kv}}{Size_{kv}} \cdot \log_T(\frac{N}{F} \cdot \frac{T-1}{T}) = T \cdot \log_T(\frac{N}{F} \cdot \frac{T-1}{T})$. While in fragmented compaction, a key-value pair will be written to each component except the last one once. In the last component, Since we allow at most M overlapped runs in each guard, a key-value pair will be rewritten up to $\frac{T}{M}$ times in the last component. Therefore, the total WA for each key-value pair is $(\log_T(\frac{N}{F} \cdot \frac{T-1}{T}) - 1) + \frac{T}{M} = \log_T(\frac{N}{F} \cdot \frac{T-1}{T}) + \frac{T-M}{M}$.

Read Amplification Read Amplification (RA) is the ratio of data size read from NVM to that from applications. In an LSM-Tree with leveled compaction, searching a key-value pair from a component only needs to check one sorted run which leads to a reading size of $\log(S) \cdot Size_{kv}$ on each component. Assume bloom filters [63] are deployed in an LSM-Tree to reduce searching overhead and a bloom filter has a false positive rate FPR. Therefore, the RA of leveled compaction will be $\log(S) \cdot \log_T(\frac{N}{F} \cdot \frac{T-1}{T}) \cdot FPR$. In fragmented compaction, we may have to check up to M overlapped runs in each component. Therefore, the total RA in the worst case is $M \cdot \log(S) \cdot \log_T(\frac{N}{F} \cdot \frac{T-1}{T}) \cdot FPR$.

Space Amplification Space Amplification (SA) is the ratio of data size stored on NVM to that from applications. Dostoevsky [59] has identified that the SA of leveled compaction will be $\frac{1}{T}$ in the worst case (i.e., key-value pairs in all lower components are updates to the key-value pairs in the biggest capacity component.). For fragmented

compaction, the worst case happens when all sorted runs in a guard includes the same key-value pairs. The SA in the worst case is M.

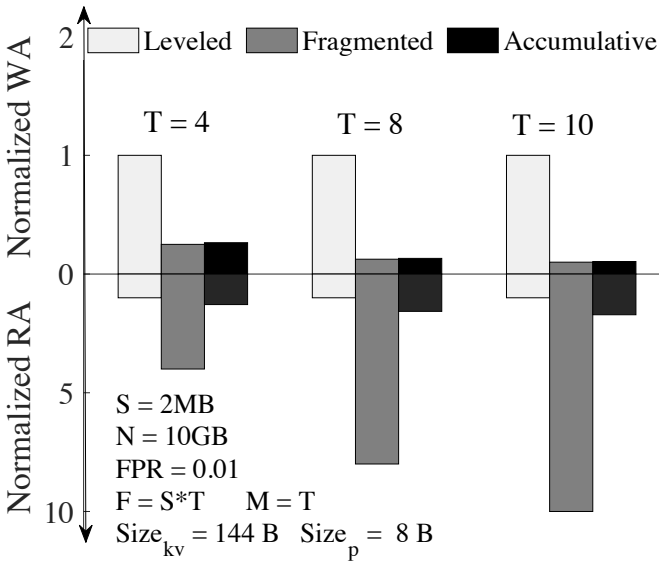


Figure 3.3: Normalized WA/RA to leveled compaction.

The leveled compaction results in a larger WA. The fragmented compaction reduces the WA significantly. While it increases the RA and SA. We also list the WA, RA and SA of the proposed accumulative compaction. The details will be discussed in Section 3.3.1. Accumulative Compaction can achieve a smaller WA and RA at the same time with a similar, but a little bit bigger SA as in fragmented compaction. Figure 3.3 shows the normalized RA and WA of different compaction schemes to those of the leveled compaction with different size ratio T. The WA of Accumulative Compaction can be as small as the fragmented compaction while its RA is closer to that of leveled compaction.

3.3 NVLSM with Accumulative Compaction

In this section, we discuss NVLSM, a design of LSM-Tree on persistent memory systems with a new accumulative compaction which is designed to fully leveraging the byte-addressability of NVM. With the accumulative compaction, NVLSM is able to reduce the write amplification without significantly increasing the read amplification.

3.3.1 Accumulative Compaction

The principal of our design is not to rewrite key-value pairs when a sorted run in a component is migrated to the next component. This is accomplished by having a pointer associated with each key-value pair and this pointer will be used to connect to the next equal or larger key-value pair in the sorted runs of the next component during "migration". We still use sorted runs of a maximum size as basic units. In a sorted run key-value pairs are sorted based on the order of their keys. In our design this process will be continued and using pointers to create multiple floors (or levels) of linked sorted runs. Please note that we can write to an empty pointer separately since this data structure is stored in NVM based memory with byte addressable capability. The linkages of these key-value pairs also enable us to efficiently search for a given key. When the number of floors in this data structure of a component reaches to a pre-determined threshold, all key-value pairs will be read out and sorted/rewrite into several sorted runs.

Figure 3.4 shows the overall design of NVLSM. As shown in Figure 3.4(a), NVLSM keeps C_0 in DRAM. New key-value pairs will be inserted and sorted in the DRAM buffer of C_0. When the DRAM buffer is filled up, the data will be flushed to C_1 directly by writing them to NVM based memory. C_1 includes a small number of sorted runs with overlapped key ranges. In each C_i ($i > 1$), NVLSM stores key-value pairs with a data structure called Skip-Array Tree (SAT). SATs in the same component have disjoint key ranges. The key range information is recorded in metadata.

An SAT organizes key-value pairs in sorted runs with multiple floors using pointers. When a sorted run is migrated to a (or next) component, it will be added to an SAT by adding a new top floor. To accelerate the searching process among the floors in an SAT, each key-value pair in the newly migrated sorted run has a next pointer pointing to the first equal or larger key in the existing floors of a sorted run. Seeking a key in an SAT is performed with a cascading searching from the top floor down to the bottom floor following the next pointers.

Figure 3.4(b) demonstrates the process of compacting key-value pairs from C_1 to C_2. C_1 consists of multiple sorted runs with overlapped key ranges. We firstly merge all key-value pairs in C_1 into a single large sorted run before partitioning them. If C_2 has no SATs (a single sorted run is also considered as a SAT), the merging results will be partitioned several sorted runs with the default size. Otherwise, the merging results are

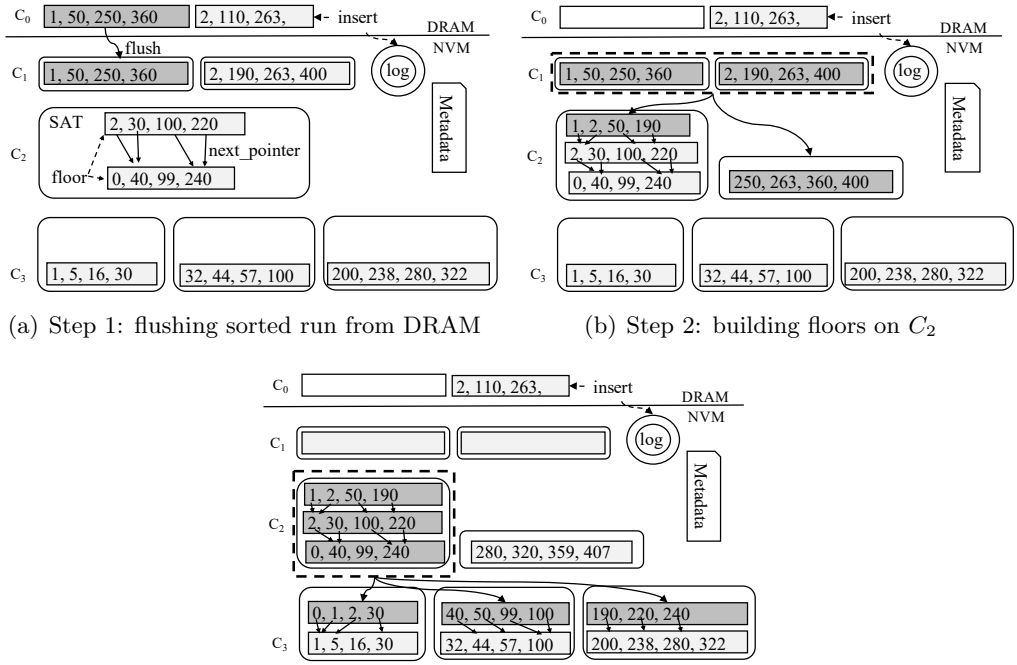
partitioned into sorted runs based on the key ranges of the SATs in C_2. Each partitioned sorted run in C_1 will be inserted into its corresponding SAT (i.e., the SAT has the same key-range) in C_2. The insertion is done by making the next pointer of each key-value pair in the sorted run pointing to a key with equal or larger value in the existing SAT. Therefore, a new top floor (sorted run) is added to the existing SAT with links to the key-value pairs in old top and lower floors of this SAT.

Even though with the help of the cascading searching following the links, searching among too many floors still decreases the search performance. Therefore, we set a maximal number of floors in each SAT. If the number of floors in an SAT reaches the maximal value, the floors in the SAT will be merged into one single large sorted run, and then partitioned and added to the top floors of SATs in the next component as we described the process for key-value pairs in C_1.

Figure 3.4(c) shows the compaction from C_2 to C_3. In the current example, we set the maximal number of floors to 3. The selected SAT will be flattened (sorted) and added to the next component as described before. If there are more than one SATs reaching to the maximal number of floors, we select the SATs with a First-In-First-Out (FIFO) policy and compact them one by one. To limit the number of the components, if NVLSM intends to compact an SAT in the last (also the largest) component, the merging results of the selected SAT will be partitioned into multiple sorted runs and written back to the last component if the last component does not exceed its maximal size based on the user-defined size ratio.

The accumulative compaction achieves a similar write amplification as fragmented compaction. It reduces the WA since the existing data in the component (except the last component) will never be rewritten during a "compaction" until the maximal number of floors is reached. At the meantime, the cascade searching on byte-addressable NVM within an SAT limits the increase of RA. We continue to discuss the WA, RA and SA of accumulative compaction using the same assumptions and metrics discussed in Section 3.2.2.

An LSM-tree includes $\boldsymbol{N}$ key-value pairs with the size ratio of adjacent components $\boldsymbol{T}$. The first component includes $\boldsymbol{F}$ key-value pairs. The total number of components is still $\log_T(\frac{N}{F} \cdot \frac{T-1}{T})$. The default size of a sorted run is $\boldsymbol{S}$ which means the average size of a single floor in an SAT is also S. We set the maximal number of floors in an SAT



(a) Step 1: flushing sorted run from DRAM

(b) Step 2: building floors on C_2

(c) Step 3: building floors on C_3

Figure 3.4: Accumulative compaction in NVLSM

to M.

Write Amplification A key-value pair will only be written once on each component except the last component. On the last component, a key-value pair may be rewritten up to $\frac{T}{M}$ times. While we also need to write a pointer together with key-value pairs. We assume the sizes of a key-value pair and a pointer are $Size_{kv}$ and $Size_p$ respectively. Therefore, the total WA will be $\frac{Size_{kv}+Size_p}{Size_{kv}} \cdot (\log_T(\frac{N}{F} \cdot \frac{T-1}{T}) + \frac{T-M}{M})$.

Read Amplification To search a key-value pair in a component, we need to check an SAT with the cascade searching. We do a binary search on the top floor first. If we assume that keys are evenly distributed, we only need to check one key for each of the subsequent floors following the links. The total reading size for searching a key-value pair in one component is $(\log_2 S + M) \cdot Size_{kv}$. We assume a bloom filter is used in each SAT whose false positive rate is FPR. Therefore the total RA is $(\log_2 S + M) \cdot \log_T(\frac{N}{F} \cdot \frac{T-1}{T}) \cdot FPR$.

Space Amplification The worst-case SA happens when all floors in an SAT include

the same key-value pairs. In this case, the real data size on NVM is $M \cdot S \cdot (Size_{kv} + Size_p)$. The data size of unique key-value pairs is $S \cdot Size_{kv}$. The total SA would be $M \cdot \frac{Size_{kv}+Size_p}{Size_{kv}}$.

The maximal number of floors in SATs (*Max_Floor*) influences the trade-off among write, read and space amplifications. When *Max_Floor* is set to 1, Accumulative Compaction becomes the same as the leveled compaction. That is, it keeps a small RA and SA while results in a large WA. As *Max_Floor* increasing, WA is reduced and both RA and SA are enlarged. However, as discussed, the increase of RA is not significant with the help of inter-floors links and the cascading searching process.

3.3.2 SAT Structure

SAT is the core data structure in Accumulative Compaction. Figure 3.5 shows the proposed SAT structure. An SAT includes an array of floor pointers (p_floor_i) (each pointer is associated with a key-value pair) which pointing to the key-value pairs in other floors (sorted runs) in this SAT, a key range *(p_range)* covered by SAT and a set of bloom filters *(p_bf)* (one for each floor). The length of the pointers will be determined by the predefined maximal number of floors. A floor consists of key-value pairs with key index and values. Each value has a corresponding index entry. An index entry can identify the offset *(val_offset)* and length *(val_len)* of a value. In current configuration, we use 4-byte *val_offset* to make sure it can index a very large sorted run. With a 2-byte *val_len*, the length of values can be up to 64KB.

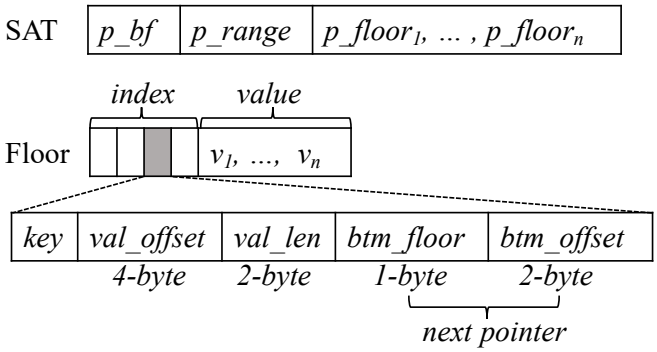


Figure 3.5: SAT structure

To achieve the cascading searching process, each index entry also includes a *next pointer* pointing to the index entry of the first equal or larger key-value pair in the lower floors. The *next pointer* has two fields, *btm_floor* and *btm_offset*. The *btm_floor* stores the position of a lower floor in the pointer array. With one-byte *btm_floor*, the maximal number of floors in an SAT can be up to 256. *btm_offset* indicates the position of the corresponding index entry in a lower floor. The 2-byte *btm_offset* can work with an index up to 65,535 entries. Note that the size of an index entry can be adjusted based on the configuration of value length, sorted run size, etc.

Adding A New Top Floor To support Accumulative Compaction, an SAT can accept new floors dynamically. Figure 3.6 demonstrates different conditions of adding a new floor to an SAT. To simplify the figure, we only show the keys in the index of each floor as a sorted run in the SAT, and use arrow lines to represent *next pointers*. When adding a sorted run to an SAT, we basically have three conditions – Overlapped (column 1: most left), Partially-overlapped (column 2: middle) and Non-overlapped (column 3) in Figure 3.6 most right.

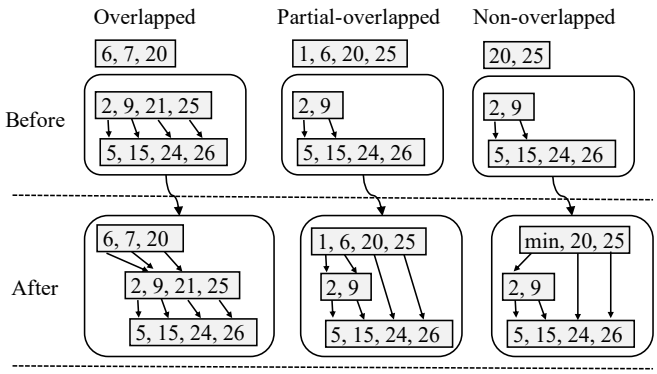


Figure 3.6: Adding a new floor to an SAT

Overlapped condition means the end key of the new sorted run is smaller than the end key of the current top floor. In the example of the overlapped condition as shown in Figure 6 column 1, we want to add a new sorted run which is $\{6, 7, 20\}$ to an SAT. The end key of the new sorted run is 20, and it is smaller than the end key 25 of the current top floor. All next pointers in the inserted sorted run will be set to point to the corresponding keys in the current top floor. A corresponding key in the current

top floor is the key which is either equal or the smallest larger one than the key in the inserted sorted run. Note that the next pointers of both 6 and 7 in the inserted sorted run will point to 9 in the current top floor since 9 is the smallest larger key than both 6 and 7. After building these links, the sorted run $\{6, 7, 20\}$ becomes the new top floor of this SAT.

In the partially-overlapped condition (Figure 3.6 column 2), the end key of the inserted sorted run is larger than the end key of the current top floor. After we go through every key in the current top floor, there are still keys in the inserted sorted run whose next pointers are empty. Therefore, we will continue go to the next bottom floor until the next point of every key in the inserted sorted run has been set or we run out the floors. For example, as shown in Figure 6 column 2 the next pointers of 1 and 6 point to 2 and 9 respectively in the current top floor, and the next pointers of 20 and 25 will point to the 24 and 26 in the bottom floor.

At last, if the inserted sorted run is non-overlapped with the existing top floor (Figure 3.6 column 3), the start key of the inserted sorted run is larger than the end key of the current top floor. If we build the links directly, the current top floor may be omitted during a cascading searching following the links. Therefore, we will add a virtual key *min* at the beginning of the inserted sorted run. The key *min* is a special number smaller than any keys in the workload. It does not influence the range of a floor and will be ignored when merging the floors in an SAT. In our example as shown in Figure 6 column 3, the next pointer of *min* will point to the start key 2 of the current top floor, and the next pointers of 20 and 25 in the inserted sorted run will point to the 24 and 26 in the bottom floor respectively. In this way, the searching process will not skip the current top floor.

Bloom filters and the key range of the floors in an SAT will also be updated after adding a new top floor. NVLSM utilizes in-place updates for bloom filters and copy-on-write for the key range. Although in-place updates may be dangerous on NVM, the compaction can tolerate the caused data inconsistency after a system failure (this will be further discussed in Section 3.3.4).

Cascading Searching Process In an LSM-Tree, searching for a target key always starts from the smallest component. The existing compaction methods reducing write amplification like tiered or fragmented compaction, result in large read amplifications

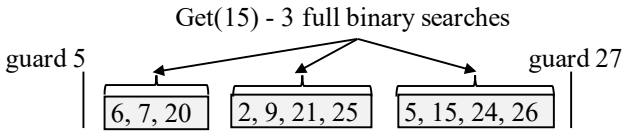


Figure 3.7: Get(15) in a guard of Fragmented Compaction

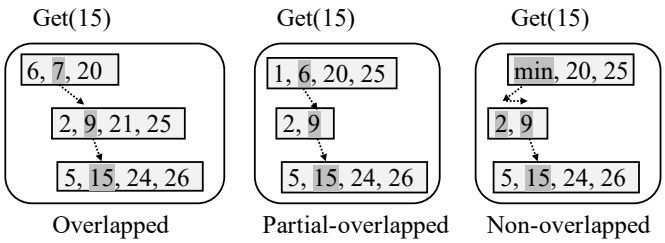


Figure 3.8: The critical path of Get(15)

since the LSM-Tree may have to check multiple sorted runs in each component. Figure 3.7 shows the process of Get(15) in a guard of fragmented compaction. When Get(15), the LSM-Tree can locate one guard in a component. However, a guard may include multiple sorted runs. In this example, the guard 27 includes 3 sorted runs. Therefore, we have to perform up to 3 full binary searches, therefore, the read amplification is increased.

NVLSM is able to reduce the write amplification while limiting the increase of read amplification. Based on the key ranges covered by SATs in a component, we can identify which SAT to be searched since the target key has to be in the key range of this SAT. By leveraging the byte-addressable data accesses of NVM, looking for a key in an SAT will be executed with a fractional cascading searching. We only need to perform a full binary search in the top floor. In the rest floors, we can read a small portion of total data with the help of the next pointers.

Figure 3.8 shows the critical path (marked with the dark color) of searching in an SAT under different conditions. Note that we only show the involved pointers. If bloom filters indicate that the target key may exist in an SAT, NVLSM will do a binary search on the top floor to locate Key_i which is the largest key smaller than the target key. If Key_{i+1} is equal to the target key, the searching will be terminated since it is the newest

version.

If Key_{i+1} is greater than the target key, the searching process will continue to search the next lower floor. The start and end positions of searching the next lower floor will be the next pointers of Key_i and Key_{i+1} respectively. For instance, in the overlapped condition, if we try to *Get(15)*, NVLSM will first conduct a binary search on the top floor and stop at the key 7 since 7 is the largest key that is smaller than 15. For the next floor, the start and end positions of searching will be key 9 and key 21 which are pointed by the next pointers of key 7 and key 20 in the top floor respectively.

If the next pointers of Key_i and Key_{i+1} point to different floors, the searching process will use the range information of corresponding floors to further identify the floor to search. In the example of partially-overlapped condition, searching on the top floor stops at the key 6. However, the next pointers of key 6 and key 20 on the top floor point to different floors. Thereby, NVLSM further checks the range of the next lower floor (the middle floor), and find that the target key 15 is not in the range $< 2, 9 >$. Therefore, the searching process will skip the middle floor following the next pointer of key 9. Finally, NVLSM searches the bottom floor between the positions of key 15 and key 24.

3.3.3 Key-Value Pair Operations

As a key-value store, NVLSM supports major key-value operations including **Put**, **Delete**, **Get**, **Seek** and **Next**.

Put/Delete. As shown in Figure 3.4, during a Put operation, a new key-value pair will be buffered and sorted in the write buffer of C_0 in DRAM and protected by a persistent log on NVM. If the write buffer is filled up, it will be flushed to C_1 on NVM directly. Thus, C_1 will have overlapped SATs with only one floor (i.e., a sorted run). After C_1 reaches to a maximal size, all data in C_1 will be merged into a single sorted run and then partitioned into multiple sorted runs with a bounded size. These sorted runs will be migrated to the next component using Accumulative Compaction. The Delete is also executed as a Put operation with a value of delete mark. The invalid key-value pairs will be dropped during future merging operations.

Get. Get operation will perform a single-point look-up. NVLSM will search the key with the order of the write buffer of C_0, every SAT in C_1 and one SAT in each other

components. As we discussed in Section 3.3.2, looking for a key in SAT will be executed with a cascading searching.

Seek and Next. The Seek and Next operations are used for range queries. A Seek operation can locate the position of the largest key which is smaller than the target key. Then a Next operation will return the next larger key. The seeking process in an SAT is also executed with a cascading searching process. However, when seeking, NVLSM will not first check the corresponding bloom filters. The Next operation in an SAT will iterate all floors at the same time and return keys in an ascending order.

3.3.4 Data Consistency and Recovery

Unexpected system shutdowns may result in data inconsistency in NVM. Therefore, NVLSM has to ensure data consistency on NVM to recover from a system failure. NVLSM utilizes the Persistent Memory Development Kit (PMDK) [64] to atomically allocate/free space from NVM devices. Specifically, NVLSM needs to ensure data consistency during two processes – writing to the persistent log for the data stored in DRAM buffer and the compaction process.

We protect the key-value pairs in the DRAM write buffer by appending them to persistent logs on NVM with a delimiter. The append operation will be followed by *clflush* and *mfence* immediately to bypass the CPU cache lines. Each write buffer has its private log which will be dropped after the data is completely flushed to NVM. When recovering after a system failure, the last key-value pair in the log for the unfilled write buffer will be ignored since it may not be flushed from cache lines. With this mechanism, NVLSM only flush cache line once for each new key-value pair inserted.

Metadata records key ranges and locations of SATs included in each component. During a compaction, Metadata will be updated. We use a similar mechanism with LevelDB, Log-And-Apply, to reduce the update overheads [65]. NVLSM keeps two versions of Metadata, a persistent version on NVM and a volatile version in DRAM. After a compaction, the volatile version of Metadata will be updated directly.

Meanwhile, we use a compaction log to record the operations during a compaction including adding new floors to SATs, updating Metadata and deleting the old data, etc. Table 3.1 shows an example of log entries for a compaction. In this compaction, NVLSM merges SAT 0 in C_1. After merging and partitioning, NVLSM generates two

sorted runs *Run* 1 and *Run* 2, and intend to add them to SAT 1 and SAT 2 in C_2. Finally, NVLSM deletes the SAT 0 after adding new runs into SAT 2. NVLSM records each operation during a compaction to avoid memory leaks, and adds a commit entry to indicate a compaction is finished.

Table 3.1: Compaction log

operation	void*, void*
start	C_1, SAT 0
add	SAT 1, *Run* 1
add	SAT 2, *Run* 2
delete	C_1, SAT 0
commit	

We use an integer variable which has a size of 4 bytes and can be in-place updated atomically to indicate the count of log entries. We reserve a space on NVM for the compaction log. When the reserved space is filled up, we create a new persistent version of Metadata by taking a snapshot of the volatile version. The old persistent version of Metadata will be deleted, and the compaction log will be cleared. After a system failure, NVLSM will redo the uncommitted compaction based on the log information. In this way, a compaction process can tolerate any temporary inconsistent state.

3.3.5 Discussions and Limitations

Key-Value Separation Separating keys from values can considerably reduce write amplification in leveled compaction since values will not be involved in the read-merge-writes [66, 67]. Although we have discussed compactions of storing keys and values together, they are compatible with key-value separation. After separating keys and values in all leveled, fragmented and accumulative compactions are still subject to the aforementioned trade-offs among write, read and space amplifications.

Component Partition To limited the transient space and time requirement for a compaction, both leveled and fragmented compactions partition components into smaller key ranges. Leveled compaction sorts key-value pairs and partitions them with small sorted runs. Fragmented compaction partitions a component using guards. Both of them can be adaptive with skewed workloads. In Accumulative Compaction, we use a similar partition method with leveled compaction. When the component is empty,

Accumulative Compaction partitions the key-value pairs from a smaller component into small sorted runs based on a maximal size of sorted runs. Then the following key-value pairs will be partitioned based on the existing data. However, Accumulative Compaction can also be compatible with other partition methods including dynamic guards.

Multi-Thread Implementation RocksDB [48] improves the compaction efficiency with a multi-thread compaction. Other studies [60, 53] deploy multi-thread seeks/reads to improve the seek/search performance. Although NVLSM currently only deploys a single thread for compaction and searches among components, it can also be further developed using multi-thread compaction and search.

Other Optimizations Cache-conscious data structures [68] can be used in memory to boost performance by increasing CPU cache line efficiency [69, 70, 71]. The current index of floors uses a plain array to store the index entries. However, it is possible to use other cache-conscious index designs for each floor. Besides, NVLSM utilizes a legacy design of DRAM component from LevelDB. It is compatible with existing studies [53, 54, 55] that optimize the DRAM component.

3.4 Related Work

In this section, we summarize some related work including persistent indexing structures, LSM-trees that can reduce write amplification, and fractional cascading researching used in our design.

Persistent Indexing Structures Consistent and durable data structures have been an essential research topic for persistent memory systems [72]. For examples, hashing index [73, 74, 75] and B+Tree [76] have been optimized for persistent memory to achieve good read performance with a small consistent overhead for updates [77, 78, 79, 80, 46]. FAST+FAIR Tree [79] is a version of B+Tree design for persistent memory systems. FAST+FAIR Tree stores both inner nodes and leaf nodes on NVM. It outperforms other persistent indexing structures by a large margin for both insert and read operations. LSM-Tree has also been optimized for persistent memory systems. NVMRocks [55] and SLM-DB [54] optimizes the LSM-Tree on systems with both NVM and storages. They cannot be applied on systems including only NVM. NoveLSM [53] redesigns LSM-Tree for NVM. It uses a mutable NVM memory table and parallel

searches to reduce the write and read latency. It still uses leveled compaction to merge sorted files. The proposed accumulative compaction in NVLSM is a complementary compaction scheme to NoveLSM and can also be applied in NoveLSM for compacting sorted files.

Reducing Write Amplification in LSM-Trees Monkey [58] and Dostoevsky [59] have analyzed the trade-offs of read, write and space overheads between leveled compaction and tiered compaction. As we discussed in Section 3.2.2, leveled compaction has smaller read and space amplifications but it considerably increases the write amplification. To reduce the write amplification of leveled compaction, existing studies separate keys from values so that values will not be involved in the read-merge-write operation [66, 67]. In Skip-Tree [81], the compaction does not have to be executed between two adjacent components. Instead, it can skip certain components, and decreases the number of read-merge-writes.

Tiered compaction is also used in many key-value stores like RocksDB [48], Cassandra [82], Scylla [57] and so on. Tiered compaction significantly reduces the write amplification but enlarges the read and space amplifications. Besides, tiered compaction requires a large transient space and time for a compaction operation [57]. Several existing studies propose other compaction methods to reduce write amplification. PebblesDB combines LSM-Tree with SkipList [83], and uses fragmented compaction to migrate data. LSM-trie [84] and SlimDB [85] divide a level (component) into multiple sub-levels to reduce write amplification, although they are inefficient for range queries since they organize key-value pairs by hashing the keys in each component. LWC-Tree [86] proposes a light-weight compaction by appending key-value pairs from a smaller component into the sorted runs of a larger component. They can achieve a write amplification close to that of tiered compaction, and requires less transient space and time overheads for a compaction. However, they still result in a large read amplification.

Fractional cascading searching Fractional cascading searching [87] is a data structure technique to reduce the searching overheads. TokuDB uses a fractal tree with cascading searching to index the stored data [88]. FD-Tree [89] and bLSM [90] use cascading searching to accelerate the searching process in FD-Tree and LSM-Tree. However, they do not target on reducing the write amplification.

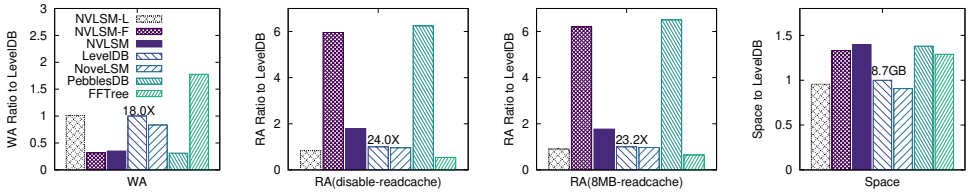


Figure 3.9: Normalized write, read and space amplification to LevelDB

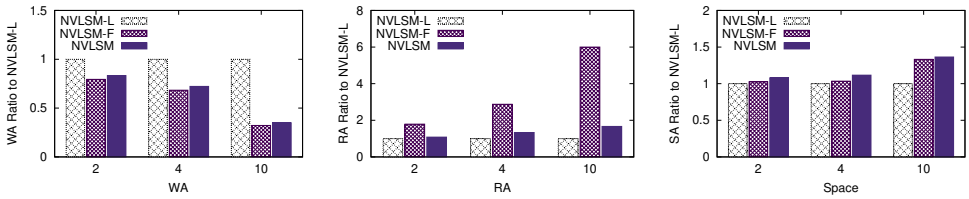


Figure 3.10: Normalized write, read and space amplification to NVLSM-L

3.5 Performance Evaluation

3.5.1 Experimental Setup

PMEMKV [91] is a key-value store implemented with Persistent Memory Development Kits (PMDK) [64]. PMDK is a collection of libraries and tools, and it can atomically allocate space from an NVM device. We implement NVLSM based on PMEMKV and emulate NVM with 56 GB NVDIMM [92]. By adding CPU cycles in write and read processes of the NVDIMM module, we approximately emulate the write/read latency of NVM devices. In our evaluations, we compare NVLSM with LSM-Tree-based key-value stores using different compaction methods. Besides, to comprehensively understand the performance of LSM-Tree on persistent memory systems, our evaluation also includes a state-of-the-art key-value store using persistent B+Tree. The configurations of key-value stores in our evaluations are shown as following.

LSM-Tree-based Key-Value Stores In our evaluation, we compare NVLSM (accumulative compaction) with two existing LSM-Tree-based key-value stores – LevelDB (leveled compaction) [47] and PebblesDB (fragmented compaction) [60]. Besides, we also setup the state-of-the-art LSM-Tree design for NVM, NoveLSM [53]. We do not compare with RocksDB since the write amplification of RocksDB is not significant

smaller than that of LevelDB when using leveled compaction [60, 93]. We do not include LSM-Trees with tiered compaction either since the write amplification of tiered compaction is close to that of fragmented compaction, but it requires a relatively larger transient space and more time for a compaction [93].

NOVA [94] is a log-structured file system for NVM systems. It achieves the best performance for append-only workloads satisfying the write behaviors of LSM-Trees. In our evaluation, we run LevelDB, PebblesDB and NoveLSM on NOVA file system with direct accesses. However, LevelDB and PebblesDB are not designed for NVM, and extra performance overheads are introduced by file system interface and data serialization. Besides, existing key-value stores, e.g. LevelDB [47], RocksDB [48], HyperLevelDB [95] and etc., apply different optimizations for many other purposes. Those optimizations may influence the performance of key-value stores to some degree but they do not change the trade-offs among write, read and space amplifications of the deployed compaction methods. For instance, multi-thread compactions in RocksDB cannot reduce the write amplification compared with a single-thread compaction in LevelDB in the existing evaluations [60, 81]. As we discussed in Section 3.3.5, the proposed accumulative compaction can also be integrated with most of the optimizations.

NoveLSM uses a mutable persistent NVM buffer and parallel searching to reduce write latency and read latency significantly. Sorted files are still used since they are efficient for range queries based on the existing evaluation [53]. The proposed accumulative compaction of NVLSM can be integrated with the mutable persistent NVM buffer and parallel searching in NoveLSM for compacting sorted files.

The current implementation of NVLSM does not include the above optimizations. Our major purpose is to show the trade-offs of different compaction schemes on write, read and space amplifications. To provide a fair comparison of different compaction schemes, we have implemented leveled compaction and fragmented compaction in NVLSM. NVLSM-L is an LSM-Tree-based key-value stores using leveled compaction implemented by PMDK based on PMEMKV, and NVLSM-F uses LSM-Tree with fragmented compaction.

Similar with [96], we deploy two threads. One is for executing benchmark and the other is for background compaction. Since most of the key-value stores use single thread to search a key, no parallel searches are applied. The number of threads will not

influence the write and read amplifications. The size of DRAM write buffer is 2MB. We disable the data compressor, and set the bloom filters to 10 bits per key. The maximal size of a sorted run is 2MB. In PebblesDB and NVLSM-F, a guard includes at most 10 sorted runs. In NVLSM and NVLSM-L, the maximal number of floors in SATs is also set to 10. In NoveLSM, the size of NVM buffer is set to about 20% of the total data size which is similar with the ratio in [53].

B+Tree-based key-value stores FAST+FAIR Tree (FFTree) [79] is a B+Tree based design for persistent memory systems. We also implement FFTree using PMDK based on PMEMKV. We set node size to 512B which is the same configuration as in [79]. The nodes of FFTree only include keys, and values are indexed by the keys of leaf nodes. We use binary search for each node since it provides a better searching performance based on our evaluation.

3.5.2 Write/Read Amplification and Space Overhead

In the first evaluation, we discuss the trade-offs among Write Amplification (WA), Read Amplification (RA) and Space Overheads (Space) in different key-value stores. In this evaluation, the key size is 16 bytes. We set the value size to 128 bytes which is the same as in [96] since the real-world workloads are dominated with small values [97, 84]. With this key/value size, a node in FFTree includes 32 keys. A sorted run in LSM-Tree-based key-value stores store about 14,400 key-value pairs. We use db_bench which is the same used in [96] to run workloads of random inserts and reads. WA and RA are measured for workloads of random inserts and reads respectively. Every workload includes 50 million operations.

To get WA and RA, we record the total size of key-value pairs written or read by key-value stores and that by db_bench respectively. The read cache may influence the RA. Therefore, we measure RAs with a read cache enabled (8MB) and disabled respectively. To get space overheads, we record the total number of sorted runs in LSM-Tree-based key-value stores, and the total number of nodes and values in FFTree. In LevelDB and PebblesDB, we only record the raw size of key-value pairs to eliminate the influence from the file system.

Figure 3.9 shows the normalized WA, RA and space overheads of the evaluated key-value stores to LevelDB. We also label the absolute values of LevelDB. Compared

with LSM-Tree-based key-value stores, FFTree has the smallest RA due to the B+Tree nature of high-degree fan-out. However, the WA of FFTree is about 1.8× higher than that of LevelDB. Since nodes of FFTree stores sorted keys, inserting to FFTree leads to a large amount of data shifts. Besides, unfilled leaf nodes increase the usage of NVM space. FFTree consumes 28.6% larger space compared with that of LevelDB.

Among all LSM-Tree-based key-value stores, we verify that NVLSM-L and NVLSM-F achieve similar WA, RA and Space with LevelDB and PebblesDB respetively. Key-value stores using leveled compaction (NVLSM-L/LevelDB/NoveLSM) achieve the small RA since they only check one sorted run in most components. However, their WA is enlarged by read-merge-writes compaction. With the help of NVM buffer, NoveLSM reduces WA by in-place updating key-value pairs by 19.8% compared with LevelDB.

Key-value stores using fragmented compaction (NVLSM-F and PebblesDB) reduce WA significantly. The WAs of NVLSM-F and PebblesDB are only about 35.4% and 33.1% of those of NVLSM-L and LevelDB respectively. However, both of them suffer from a large RA since they have to check up to 10 sorted runs in each component. The RAs of NVLSM-F and PebblesDB are 5.7× and 6.1× larger than those of NVLSM-L and LevelDB respectively. Since multiple versions of data may exist in different sorted runs in the same component, NVLSM-F and PebblesDB occupy about 37.6% and 38.8% more space than NVLSM-L and LevelDB. Although a read cache can reduce the RA, it cannot change the trade-offs among read, write and space amplifications in a compaction. In the rest of our evaluations, we keep an 8MB (defaul size in DB_Bench) read cache.

NVLSM realizes a smaller WA and RA at the same time comparing with key-value stores with leveled compaction or fragmented compaction. With 128-byte values, NVLSM reduces the WA by up to 67% compared with NVLSM-L and LevelDB, and the WA of NVLSM is close to those of NVLSM-F and PebblesDB. Meanwhile, the RA of NVLSM is smaller than those of NVLSM-F and PebblesDB by 69.5% and 71.1% respectively. As we discussed, the NVM buffer in NoveLSM can be applied to both fragmented compaction and accumulative compaction to further reduce WA. NVLSM has the largest space overhead since the inter-floor pointers introduce extra space overhead. The increase of SA is not significant with value size of 128 bytes, i.e. it increases about 2.6% compared with that of NVLSM-F.

The maximal number of floors in an SAT (*Max_Floor*) influences the trade-offs

among WA, RA and space overhead. More floors can reduce WA more significantly. However, it may lead to a larger RA and space overhead. Figure 3.10 shows the normalized WA, RA and Space of NVLSM and NVLSM-F with different *Max_Floor* (x-axis) to NVLSM-L. In this experiment, we keep the same configuration and vary *Max_Floor* with 2/4/10 (NVLSM-2/4/10). The WA and RA are still measured during the workloads of Put and Get respectively.

Figure 3.10 shows the impact on the trade-offs among WA, RA and SA from *Max_Floor*. Results indicate that a larger *Max_Floor* can help to reduce WA more significantly. However, it also increases RA and Space overhead to a lesser degree. When *Max_Floor* is 2, NVLSM-2 reduces WA by 15% compared with NVLSM-L. When *Max_Floor* is increased to 10, NVLSM-10 can reduce WA by 65% compare with NVLSM-L.

Although RAs of both NVLSM and NVLSM-F are increasing with the increase of *Max_Floor*, by comparing RAs of NVLSM and NVLSM-F, we find that the inter-floor links and the cascading searching process in an SAT mitigate the RA increase significantly. When *Max_Floor* is 10, the RA of NVLSM-F is enlarged by 6× compared with that of NVLSM-L. With the help of the cascading searching, the RA of NVLSM-10 is only 1.67× of NVLSM-L.

The multi-floor SAT increases space overhead since old versions of key-value pairs may co-exist in different floors. A larger *Max_Floor* will lead to a slightly higher space overhead. The space overhead of NVLSM-2 is only about 1.1× of NVLSM-L. When *Max_Floor* is increased to 10, the space overhead of NVLSM-10 is about 1.42X of NVLSM-L. By comparing NVLSM with NVLSM-F, we find that the inter-floor links of SATs do not introduce significant space overhead. The space overhead of NVLSM-10 is about 2.7% higher than that of NVLSM-F.

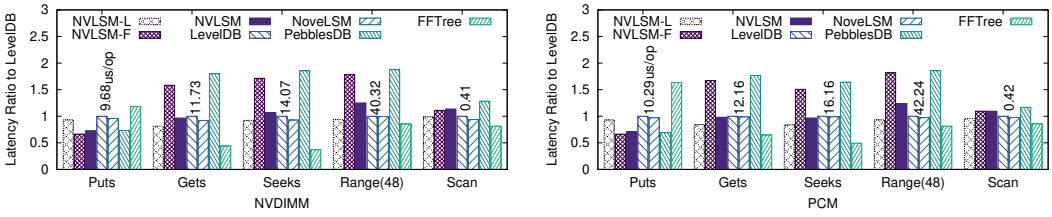


Figure 3.11: Latency comparisons with 128-byte values

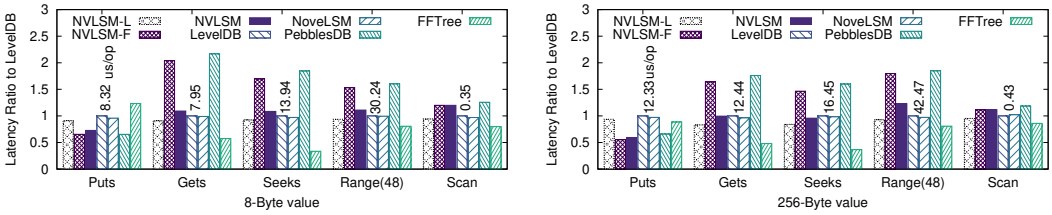


Figure 3.12: Latency comparisons with variable value sizes

3.5.3 Latency Comparison

We run 5 workloads including Puts, Gets, Seeks, Range and Scan, using db_bench. Puts/Gets/Seeks represent putting/getting/seeking key-value pairs with a random order. Range(48) stands for small range queries. In a Range(48) operation, we firstly call seek() for a random key and then call next() 48 times. Scan means we scan the whole data set with an ascending order.

In the first evaluation, we uses the same configuration as in Section 3.5.2. We also emulate different types of NVM by varying the delay of the read and write operations. Assume that the read and write latency in DRAM is 50ns [24, 25]. We emulate two types of NVM. NVDIMM is battery-backed DRAM [92]. It represents the NVM that provides similar read and write latencies as DRAM. Phase Change Memory (PCM) [98] stands for the NVM which achieves a larger capacity, but is slower than DRAM. In our evaluations, we set the read/write latency of PCM to 100ns/400ns respectively [25, 94].

Figure 3.11 shows the results of latency (μs/operation) comparisons with different types of NVM. We show the normalized latencies to LevelDB, and label the absolute latency of LevelDB. We discuss the results in the following aspects.

File System Overhead LevelDB and PebblesDB use file system interfaces. Data serializations have to be performed due to the inconsistent formats between persistent files and in-memory data structure. NVLSM-L and NVLSM-F manage and operate NVM device directly bypassing the file system. They use a unified data structure for both NVM and DRAM to eliminate the data serializations. By comparing NVLSM-L and NVLSM-F with LevelDB and PebblesDB correspondingly, we identify that NVLSM-L and NVLSM-F achieve 10%-15% performance improvements in both Put and Get workloads.

LSM-Tree and FFTree LSM-Tree based key-value stores can provide better Put performance than FFTree since inserting in FFTree leads to a large amount of data movements. Besides, in FFTree, a search operation has to be performed before each insert. This also introduces considerable latency. The Put latency of LevelDB is 23.4% – 25.7% smaller than that of FFTree. However, FFTree outperforms LSM-Tree-based key-value stores in all other read-intensive workloads, including Get, Seek, Range and Scan.

Random Puts Among LSM-Tree-based key-value stores, NVLSM offers a better insert performance than those with leveled compaction (LevelDB, NVLSM-L and NoveLSM) since accumulative compaction reduces write amplification significantly. Compared with LevelDB and NVLSM-L, NoveLSM reduces the insert latency by 8.8% – 9.1%, and NVLSM reduces the insert latency by 32.1% – 37.7% and 24.4% – 30.4% respectively. The Put latency of NVLSM is close to that of PebblesDB and NVLSM-F. Shorter insert latency can be achieved if we incorporate NVM buffer with fragmented/accumulative compaction. By comparing NVLSM with NVLSM-F, we identify that accumulative compaction introduces about 1.3% – 2.2% more overhead than that of fragmented compaction.

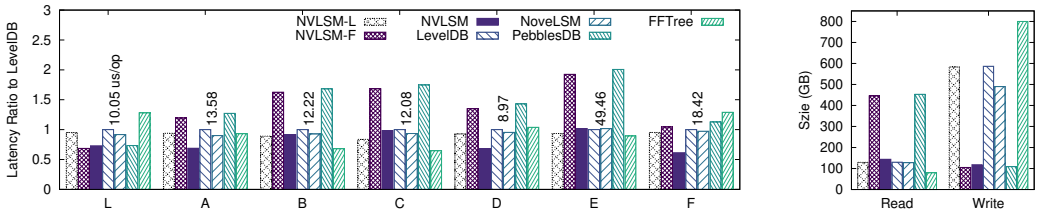
Random Gets With the help of the inter-floor links and the cascading searching, NVLSM reduces the Get latency by 45.6% – 51.3% compared with those of PebblesDB and NVLSM-F. However, the Get latency of NVLSM is larger than that of NVLSM-L by 11.9% – 17.3%.

Random Seeks Compared with PebblesDB and NVLSM-F, the Seek latency of NVLSM is reduced by 43% – 48% since the seeking process in NVLSM can also be accelerated by the cascading searching. NVLSM-L provides the best seeking performance since it seeks only one sorted run in each component. The seeking latency of NVLSM is larger than that of NVLSM-L by 11.2% – 15.6%.

Small Ranges In small range queries, Range(48), the total latency is influenced by the seeking latency significantly. Thereby NVLSM can still provide a good performance close to that of LevelDB since the seeking process is accelerated with the cascading searching. However, NVLSM has to iterate more sorted runs than LeveDB and NVLSM-L. The total latency of NVLSM for Range(48) is enlarged by 23.5% – 26.8% compared with NVLSM-L. While PebblesDB and NVLSM-F result in a large latency for small

range queries due to their seeking overhead. Compared with NVLSM-F, the small range query latency of NVLSM is reduced by 33.4% – 37.7%.

Scan When scanning the data, the latency is dominated by the iteration process for searching key-value pairs instead of the seeking process. In this case, NVLSM and NVLSM-F have a similar performance. Their scan latency is larger than that of NVLSM-L by 15.3% – 17.2% since they have to iterate key-value pairs among more sorted runs at the same time.



(a) Normalized Latency to LevelDB

(b) Write/Read Sizes on NVM

Figure 3.13: Evaluations with Real-World Workloads

Latency Sensitivity From the results of NVDIMM and PCM in Figure 3.11, we can find that when the latency of NVM increases, the improvements of the Put performance become larger. With NVDIMM, the Put latency of NVLSM is 24.4% shorter than that of NVLSM-L. With PCM, the latency reduction of NVLSM to NVLSM-L becomes 30.4%. However, the latency for Get is not increased significantly like in PebblesDB or NVLSM-F. From NVDIMM to PCM, the increase of get latency changes from 11.9% to 17.3%. This is because NVLSM reduces the write amplification significantly and only less significantly increases the read amplification with the help of the cascading searching.

Variable Value Sizes In this evaluation, we study the impact on the operation latency from different value sizes. Besides 128 bytes, we increase the value size to 256 bytes. We also include an extremely small value size, 8 bytes. The extremely small value can be used in some special scenarios like index server [62]. It can also represent the key-value separation in some degree since keys will only record the addresses of values with several bytes [66]. The results in Figure 3.12 indicate that NVLSM can improve the Put performance for both smaller and larger value sizes. NVLSM can

achieve a better performance improvement with larger value sizes. In the results of 256-byte value, compared with LevelDB, NVLSM reduces Put latency by 47.6% with a comparable Get latency.

3.5.4 Other System Overheads

DRAM Space and CPU Utilization We randomly insert 100M key-value pairs with key/value size of 16/128 bytes, and then perform 100M Get operations. We measure the DRAM space and CPU utilization during the execution. We check the CPU utilization every 10s. The median CPU utilization of NVLSM-L, NVLSM-F, NVLSM, LevelDB, NoveLSM, PebblesDB and FFTree are 132%, 155%, 140%, 138%, 140%, 157% and 159% respectively. NVLSM has a lower CPU utilization than those of PebblesDB and NVLSM-F since it reduces the read amplification during the searching.

LSM-Tree-based key-value stores need a small amount of DRAM for the write buffer and metadata. The DRAM space used by NVLSM-L, NVLSM-F and NVLSM is about 8MB. LevelDB and NoveLSM consume about 9MB – 10MB DRAM. Although PebblesDB requires a large DRAM space since it keeps bloom filters in DRAM, it can achieve similar amount of DRAM with LevelDB and NVLSM if it stores bloom filters on NVM.

Crash and Recovery We also measure the recovery time of FFTree, LevelDB, NVLSM and PebblesDB for recovering 100M key-value pairs with key/value size of 16/128 bytes. All of the key-value stores can achieve a fast recovery. The recovery time are all under 1 second. Although PebblesDB can also be reopened within a second after a failure, it has to reconstruct bloom filters when sorted runs are accessed at the first time.

3.5.5 Real-World Workloads

Yahoo! Cloud Service Benchmark (YCSB) provides 6 core workloads representing different real-world scenarios [99, 100] including **L** – 100% Puts, **A** – 50% Puts, 50% Gets, **B** – 5% Puts, 95% Gets, **C** – 100% Gets, **D** – 5% Puts, 95% Gets (Latest), **E** – 5% Puts, 95% Short Ranges(100) and **F** – 50% Read-Modify-Writes, 50% Gets. We still use key/value size of 16/128 bytes and use the same configuration in Section 3.5.2. We

randomly load 100M key-value pairs (L) before executing each workload (A – F). We emulate NVM performance with PCM and perform 10M operations for Workload A – F.

Figure 3.13 shows the normalized latencies to LevelDB (Figure 3.13(a)) and the total write and read sizes of NVM for all workloads (Figure 3.13(b)). In the results of Figure 3.13(a), FFTree provides a good performance in read-intensive workloads, e.g. B and C. LSM-Tree-based key-value stores achieve a small latency in write-intensive workloads.

Among all LSM-Tree-based key-value stores, the results of Workload L indicate NVLSM-F/PebblesDB can provide good Put performance. However, their performance is degraded by the long Get latency under Workload A – F. NVLSM offers a good performance in L, A, and F Workloads since it achieves better Put performance similar with PebblesDB and a Get performance comparable with LevelDB at the same time. In the Read-Modify-Write workload (F), the latency of NVLSM is smaller than that of key-value stores with other compaction schemes by 48.7% – 61.1%. Correspondingly, NVLSM achieves about 1.92× – 2.56× higher throughput (ops/second) than other key-value stores.

Figure 3.13(b) shows the total write and read sizes for all 7 workloads. NVLSM is the only key-value store that reads and writes a small amount of data at the same time. Compared with PebblesDB and NVLSM-F, NVLSM reduces the read size by 63.1% – 64.5% and achieves a similar read size to that of LevelDB and NVLSM-L. Meanwhile, NVLSM also achieves a small write size close to that of PebblesDB and NVLSM-F. Compared with LevelDB and NVLSM-L, NVLSM reduces the write size by 82.7% – 83.3%.

3.6 Conclusion

We propose NVLSM using LSM-Tree with a new accumulative compaction scheme. Accumulative compaction fully utilizes byte-addressability of NVM to build floors in SATs when migrating sorted runs between components, and perform an efficient cascading searching among floors. Our evaluations show that NVLSM reduces write amplification without significantly increasing read amplification. In our future work, we would like to improve the current design by involving both NVM and storage.

Chapter 4

Idler: I/O Workload Controlling for Better Responsiveness on Host-Aware Shingle Magnetic Recording Drives

4.1 Introduction

Shingled Magnetic Recording (SMR) is a technology to increase the areal density of spinning drives by overlapping data tracks [101]. In an SMR drive, overlapped data tracks are grouped into zones, and each zone has a write pointer to indicate the current write location. An update is considered a non-sequential write if it does not begin at the write pointer in a zone. A non-sequential write may destroy the existing content in its adjacent tracks if it writes directly in its location. To avoid impacting the overlapped data, Host-Managed SMR (HM-SMR) drives prohibit non-sequential writes. That means existing applications need to be modified if we intend to use HM-SMR drives or a log-structured file needs to be used to support applications.

Device-Managed SMR (DM-SMR) and Host-Aware SMR (HA-SMR) drives implement a Shingled Translation Layer (STL) to handle non-sequential updates. STL includes an on-device persistent cache called Media Cache and an Extent Mapping Table.

A non-sequential write will be buffered in the media cache first, and later migrated to its designated location by a cleaning process. Data can be either stored at the media cache or its designated location. Any data access needs to check with the mapping table to identify its current location. The cleaning process can be triggered by either an idle duration (i.e., a duration without any requests) called idle cleaning, or by depleted STL resources (e.g., media cache free space or free entry slots in the mapping table) called blocking cleaning. HM-SMR drives can only be used in workloads without any non-sequential writes like data archives and backups or with the support of a log-structured file. DM-SMR and HA-SMR drives can be deployed in many more scenarios since they both can handle non-sequential writes with the help of STL [14].

In the modern storage infrastructure, the tail response time of a system becomes significant. Applications require a low and predictable tail I/O response time [102, 103, 104, 105]. The 99.9th percentile (P99.9) response time is used as a useful statistical metric to evaluate the tail response time [105, 106, 103, 104, 107, 108]. However, HA/DM-SMR drives suffer from a long tail response time caused by the blocking cleaning. Evaluations [109, 37] show that during a blocking cleaning, an SMR drive spends up to seconds to serve a single I/O request. Although workloads has burstiness, an SMR drive cannot rely on idle cleaning only to clean the media cache. An idle cleaning requires a relatively long idle duration to trigger and will be interrupted by any newly arrived requests. Therefore, the data cleaning efficiency of idle cleaning is degraded and blocking cleaning is still triggered. Moreover, the beginning time and the lasting duration of a blocking cleaning are unpredictable. The unpredictable performance with a long tail response time limits the usage of SMR drives. Therefore, it is critical to have a better understanding of the cleaning process. Some researchers have already conducted studies in [109, 37].

In this paper, we further evaluate the cleaning process of HA-SMR drives and explore ways of mitigating the caused performance degradation. We choose HA-SMR drives to evaluate since HA-SMR drives provide interfaces with Zoned Block Command (ZBC) [36] to check the current states of its internal structures. The internal information is helpful for our design and evaluations. By conducting evaluations, we find an idle cleaning triggered by a minimal idle duration has limited impacts on the drive performance since it stops cleaning once new requests are issued. While a blocking

cleaning considerably degrades the drive performance and increases the I/O response time significantly.

Inspired by the different influences on the drive performance from idle cleanings and blocking cleanings, we propose a scheme called **Idler** to decrease the long tail response time of an HA-SMR drive caused by the blocking cleaning. Idler considers both the resource utilization of the media cache and the current workload characteristics to induce idle durations adaptively. With the induced idle duration, data in the media cache can be cleaned in a controllable way by idle cleanings without increasing the I/O response time significantly. It can also be implemented in the firmware of DM-SMR drives. We evaluate Idler with real-world workloads. Our evaluations show that Idler can avoid blocking cleanings in workloads with a non-sequential write ratio of 10%. The tail response time and the workload finish time are reduced by 56% – 88% and 10% – 23% respectively compared with those without such control. With the help of an external write buffer on an SSD, the tail response time of Idler can be further reduced. However, the performance of Idler is still worse than that of a conventional disk drive.

In the rest parts of the paper, Section 4.2 evaluates the data cleaning process. Section 4.3 discusses the performance improvements of artificially triggered idle cleanings. Then we present the proposed Idler in Section 4.4. We evaluate Idler in Section 4.5. Section 4.6 discusses the existing work. Section 4.7 offers conclusions and the future work.

4.2 Media Cache Cleaning and Evaluation

We evaluate the media cache cleaning using HA-SMR drives whose internal structure is shown in Figure 4.1. It includes STL and SMR zones. An SMR zone is a collection of overlapped data tracks [36]. STL consists of a media cache and a mapping table. The buffered data in the media cache are log-structured since the media cache also consists of shingled tracks. Each shingled zone has a writing position called a write pointer. Any write beginning with the position pointed by the write pointer is considered as a *Sequential SMR Write* and can be issued directly to its designated location. The position of the write pointer will also be updated accordingly after a sequential write. A write with its targeted address not beginning at the write pointer is called a *Non-Sequential SMR Write*. A non-sequential write will be re-directed to media cache by

STL. Its designated zone will be changed from a sequential state to a non-sequential state. The write pointer is no longer valid in a non-sequential zone. All the following write requests to a non-sequential zone will be buffered in the media cache [37].

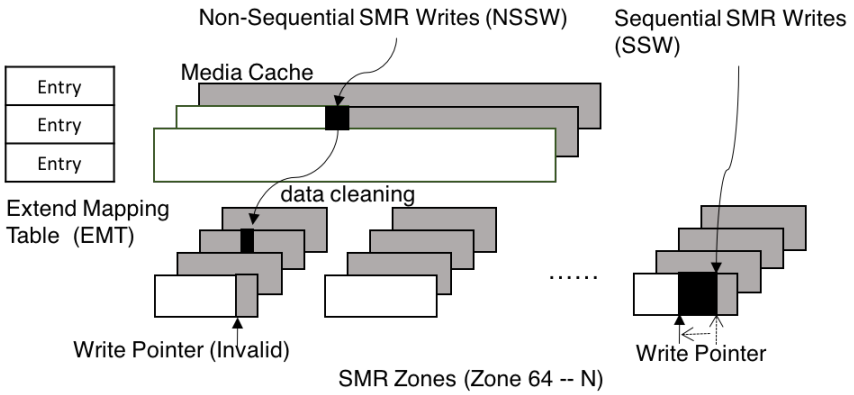


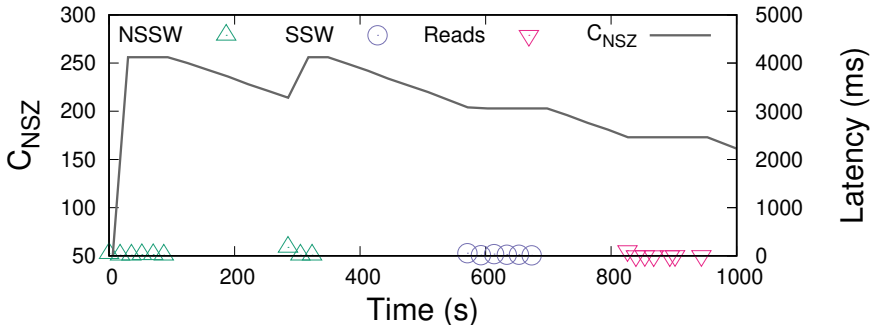
Figure 4.1: Internal Structure of HA-SMR Drives

STL utilizes a cleaning process to migrate the buffered data from the media cache to its designated locations in a Read-Modify-Write fashion [109]. A zone will be switched back to a sequential state after being cleaned. The cleaning process will be triggered once one of the resources (either free space in the media cache or free entry slots in the mapping table) is depleted [109].

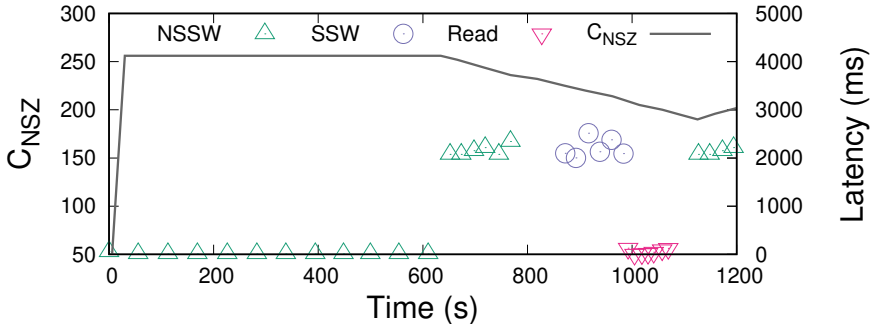
In the rest of this section, we investigate how both the blocking cleaning and the idle cleaning affect the drive performance. We use *fio* to benchmark a raw HA-SMR drive (Seagate ST8000AS0022, 8TB Capacity) with direct I/O. In this drive, the size of the media cache is 28GB, and the maximum number of mapping entries is 182,000.

Figure 4.2 shows the influences on the I/O response time by idle and blocking cleanings respectively. In this experiment, we use a single I/O thread. Since an ongoing cleaning increases the I/O response time and decreases the count of non-sequential zones, we use the change of the I/O response time (right y-axis) and the count of non-sequential zones (C_{NSZ}, left y-axis) to indicate when the cleaning starts. C_{NSZ} can be checked by Zoned Block Commands [36].

Figure 4.2(a) shows the impact of idle cleanings on the I/O response time. We issue a set of non-sequential SMR writes (NSSWs) and then idle for a duration. Once we detect an idle cleaning starts, i.e., the number of non-sequential zone decreases at



(a) Idle Cleaning



(b) Blocking Cleaning

Figure 4.2: Service Latencies of an HA-SMR Drive

around 100s on x-axis, we issue more requests including non-sequential SMR writes (NSSWs), sequential SMR writes (SSWs), and reads. The I/O response time of the following requests is not increased. It indicates an idle cleaning has limited impacts on the I/O response time since it will stop if any I/O requests are issued. For the blocking cleaning (Figure 4.2(b)), we issue a set of NSSWs to trigger a blocking cleaning. At about 650s, a blocking cleaning is triggered since the I/O response time is increased. Then we issue more SSWs, Reads, and NSSWs. The response time of the following requests is increased due to blocking cleaning. The response time of all requests after a blocking cleaning triggered is longer than that in Figure 4.2(a).

I/O dependencies influence the I/O response time since an I/O request may wait in the I/O queue before submitted to the device. When there is only 1 I/O thread, i.e., most of the I/O requests are dependent, subsequent requests may need the results of the previous requests. The waiting time in the I/O queue will be short because a new

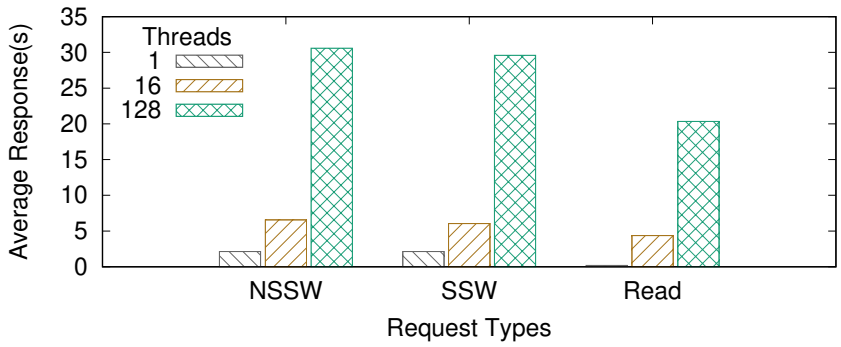


Figure 4.3: The Response Time with IO Dependencies

I/O request is only generated after the last request finishes. However, the drive may have multiple I/O threads. I/O requests from various applications are independent and submitted to the I/O queue concurrently. An SMR drive has only one disk head to serve I/O requests one by one. The I/O wait time of concurrent I/O requests will be increased since a request has to wait until the drive finishes all of the previous requests. If a blocking cleaning is ongoing on an SMR drive, the I/O wait time of a later request will be increased significantly due to the enlarged I/O response time of the previous requests. Therefore, in the next experiment, we study the influences from a blocking cleaning on the I/O response time under different I/O dependencies.

Figure 4.3 shows the average I/O response time under different I/O dependencies when a blocking cleaning is ongoing. We emulate the I/O dependencies by issuing requests with varying numbers of threads. We assume only the I/O requests in the same thread are dependent. In this experiment, we firstly trigger a blocking cleaning by issuing enough non-sequential writes. Then we start different numbers of threads to issue I/O requests including non-sequential SMR writes (NSSW), sequential SMR writes (SSWs) and reads. From the results, we find that when the number of threads increases, the I/O response time are enlarged more severely. The reason is that with more independent threads, more I/O requests submitted to I/O queue concurrently. With the increased response time of previous requests, the wait time of the later I/O requests are largely increased. As a result, the total I/O response time will be increased to an extremely large value with multiple independent threads due to the enlarged I/O wait time.

Table 4.1: Time Durations to Trigger an Idle Cleaning

IO size	Idle Time	Non-sequential Zone
4KB	240ms	8, 8, 8
	245ms	8, 8, 8
	250ms	7, 6, 6
	260ms	0, 0, 0

An idle cleaning utilizes the idle duration to clean the media cache. It is critical to find its minimum trigger time. In this experiment, we choose 8 zones and issue one non-sequential write to each with a given idle time between two subsequent requests. After 8 non-sequential writes, we check the number of non-sequential zones right away. We repeat the experiment three times for the same idle duration.

Table 4.1 shows the number of non-sequential zones. It should be 8 if an idle cleaning is not triggered by this idle time. We only show the results of 4KB IO size here even though we perform the experiments with 4KB, 64KB, 256KB, 512KB, and 1MB I/O sizes since they have similar results. When the idle time reaches 250ms, the number of non-sequential zones is less than 8 in all tests indicating an idle cleaning is triggered by a 250ms idle time. Note that it takes some time to clean even one zone. The real minimum triggering time may be a bit shorter than 250ms.

4.3 Improving Drive Performance with Artificially Triggered Idle Cleanings (AT-IC)

4.3.1 AT-IC

A blocking cleaning can largely increase the I/O response time. Although an idle cleaning has less impact on drive performance, it needs some minimal trigger time and stops with any newly arrived I/O requests. Idle cleanings may work well in a workload with burst I/O requests and relatively long idle time between bursts. However, in a workload with a short idle time smaller than minimal triggered time, idle cleanings might not even be triggered.

Inspired by the different performance influences from idle cleanings and blocking cleanings, we explore a potential concept of reducing the large tail response time caused

by blocking cleanings using Artificially Trigger Idle Cleanings (AT-IC). AT-IC mitigates the performance degradation caused by a blocking cleaning with a way of invoking idle cleanings. AT-IC artificially creates idle durations when executing a workload. The I/O requests issued during an induced idle duration are stalled and served in the next execution duration. With these artificially created idle durations, idle cleanings are invoked to conduct cleaning and will not be interrupted for a fixed idle duration. With the invoked idle cleanings, a drive can clean the data in the media cache in a controllable manner which does not significantly enlarge the I/O response time.

The benefits of AT-IC is highly related to idle configurations and the characteristics of a workload. For a given workload, two configuration parameters, idle ratio and idle length, have the most significant impacts. The idle ratio means the proportion of total idle time in the duration of executing the whole workload. The idle length means the duration of each idle period. A higher idle ratio means we create longer idle time such that an idle cleaning can clean more buffered data. However, it also means that the total execution time will be longer. If the idle ratio is fixed, an idle cleaning can be triggered frequently with a smaller idle length or less frequently with a larger idle duration. Since I/O requests are delayed during an idle duration, with a shorter and frequent idle length, I/O requests will have a shorter response time. However, triggering idle cleanings more frequently with a smaller idle duration is inefficient since an idle cleaning needs some minimal idle duration, 250ms based on our previous evaluations, to trigger.

Besides, characteristics of a workload including request arrival speeds, non-sequential write ratios, I/O request sizes, and the number of non-sequential zones may also influence the performance of AT-IC. The request arrival speed represents the intensity of a workload. It determines the number of requests delayed during an idle duration. A higher arrival speed makes AT-IC delays more I/O requests during a given idle duration leading to a larger I/O response time. The non-sequential write ratio represents the consuming speed of media cache resources. A higher non-sequential write ratio means we may fill up the media cache more frequently. The I/O request size may also influence the time duration to fill up the media cache. With a larger I/O request size, the media cache is filled up in a shorter duration, and blocking cleaning may be triggered by the depleted free space. While with a smaller I/O request size and a lower non-sequential write ratio, the media cache can serve for a longer time before a blocking cleaning is

triggered. However, a high non-sequential write ratio and a smaller I/O request size may trigger a blocking cleaning due to the depleted mapping entries. At last, a lower number of non-sequential zones represents a shorter time to clean the media cache by either blocking cleanings or idle cleanings.

AT-IC can reduce the performance degradation caused by blocking cleanings. It makes the I/O response time more predictable and provides much better performance than that without such control. However, all of the above factors influence the effects of AT-IC. Therefore in the next subsection, we validate the performance improvement achieved by AT-IC and discuss the impacts from the idle configuration and workload characteristics.

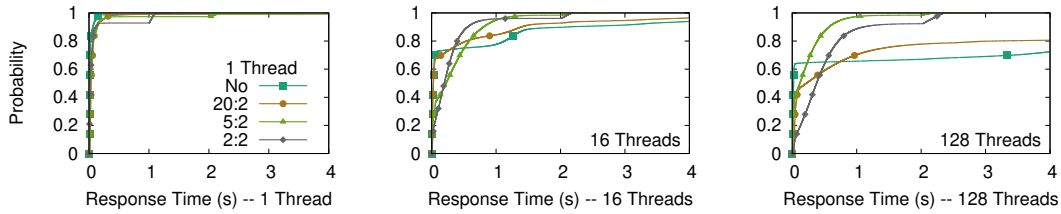


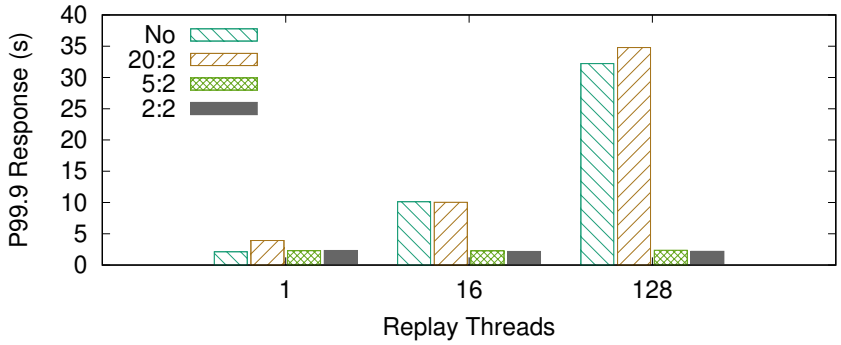
Figure 4.4: CDF of the response time with different **idle ratios**

4.3.2 AT-IC Validation

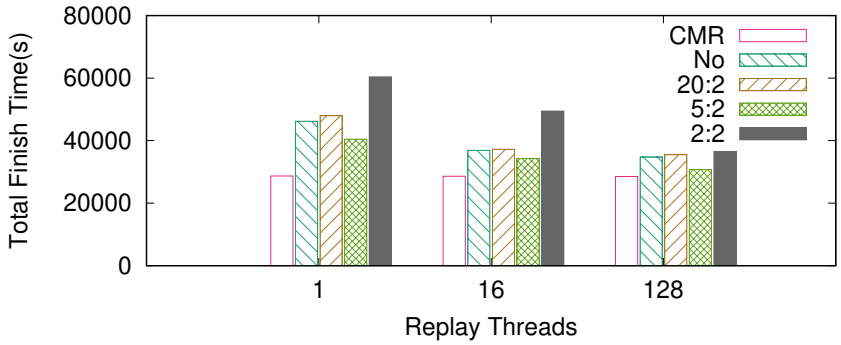
We implement AT-IC as a block I/O replay engine with libaio [110]. AT-IC issues block I/O requests respecting the time intervals between I/O requests. Beside, AT-IC can also periodically add a user-defined idle duration.

Idle Configurations

Since SMR drives are generally deployed with workloads of low non-sequential write ratios, we first evaluate AT-IC with workloads of low ratios of non-sequential writes. We vary the two parameters in the idle configuration: idle ratios and idle lengths. We synthesize traces consisting of I/O requests of 1MB size. Here we use a large I/O size since a larger request size fills up the media cache sooner. And it will negatively influence the performance of AT-IC. However, it has less impact on the total cleaning time since it is the distance between the update offset and the write pointer position



(a) P99.9 response time



(b) Total finish time

Figure 4.5: P99.9 response time and total finish time of different **idle ratios**

that dominates the cleaning time [37]. The request intervals are evenly distributed in the range from 10ms to 100ms. 10% of the requests are non-sequential writes with a total of 35GB non-sequential writing size that spread over 1024 zones. Another 10% of requests are sequential writes, and the remaining 80% of requests are reads. To study the conditions of different I/O dependencies, we partition the same trace into different numbers of subtraces (1, 16, 128). The I/O requests will be distributed evenly among subtraces based on their timestamps. Finally, we replay every subtrace with a separate thread. Only the requests within the same thread are dependent.

Idle Ratio In this experiment, we intend to discover the influence of various idle ratios on the performance of AT-IC. AT-IC replays a workload with different idle ratios while keeping the same idle length. Here we set the idle length to 2s since it is close to the largest service latency when a blocking cleaning is ongoing on a drive. Figure

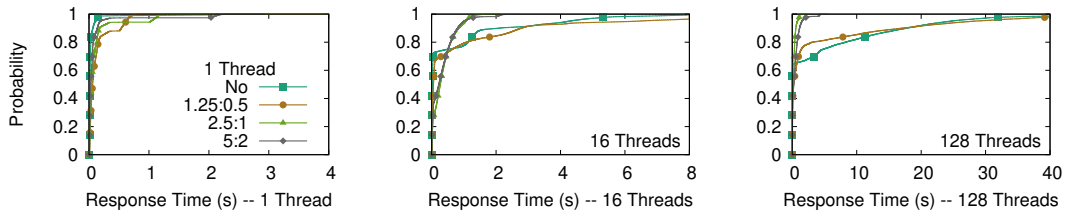
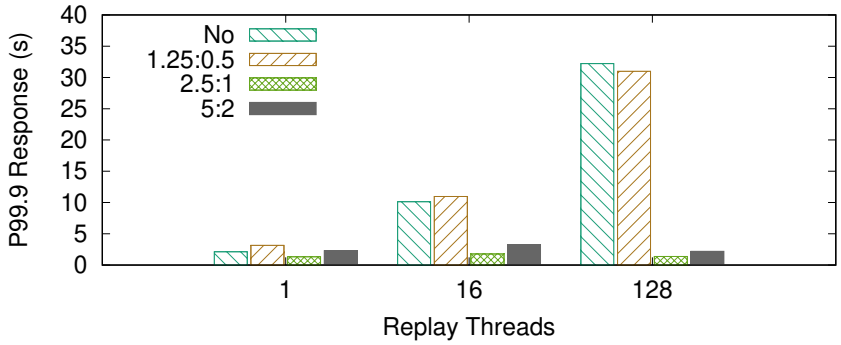


Figure 4.6: CDF of the response time with different **idle lengths**

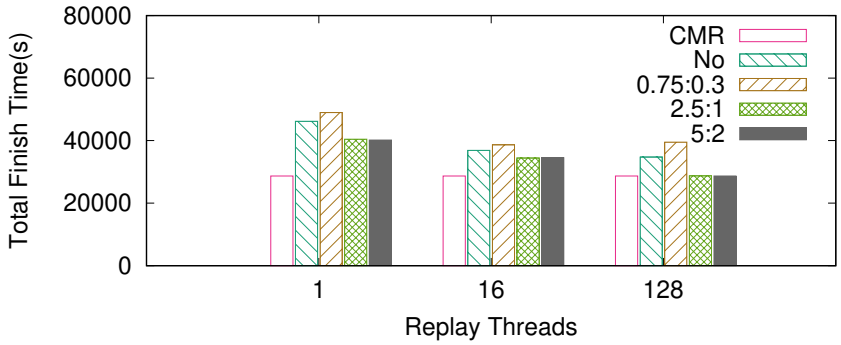
4.4 shows the Cumulative Distribution Function (CDF) probability of the I/O response time with different idle ratios. The line marked with "No" means we replay the trace with an idle ratio of 0%. The others marked with the $Execution : Idle$ like $< 20 : 2 >$ means the drive serves requests for 20s and then idle for 2s and the idle ratio is about 2/22=9.09%. We measure I/O response time with different idle ratios under different numbers of I/O threads (1, 16, 128).

With only 1 thread (Figure 4.4), all requests are dependent. The distribution of the response time of AT-IC is similar with that of "No". The reason is that the idle length, 2s, is close to the largest response time when a blocking cleaning is ongoing. The current idle length blocks the subsequent request for a time duration same with blocking cleanings. Better results can be achieved with a shorter idle length which will be discussed later. However, the P99.9 response time with 1 thread in Figure 4.5(a) shows that with a too-small idle ratio, .e.g $20 : 2$, AT-IC even increases the tail response time since it can not avoid blocking cleanings and introduces extra delay by the idle duration.

When the number of I/O threads increases (Column 2 and 3 in Figure 4.4), i.e., more independent requests are issued concurrently, AT-IC can achieve a better tail response time than no controls. The P99.9 response time of AT-IC with $< 5 : 2 >$ and $< 2 : 2 >$ is reduced by over 90% with 16 and 128 threads respectively compared to the results of no controls since blocking cleanings are not triggered when using AT-IC. Even though the concurrent requests wait in the I/O queue during an idle duration, they are satisfied in a short time after the idle duration. While if a blocking cleaning is triggered, the drive spends up to seconds to serve a request. The I/O response time will be increased to a huge value due to the extended waiting time, as we discussed in Section 4.2. The results of $< 20 : 2 >$ shows that with a too-small idle ratio, AT-IC only has limited



(a) P99.9 response time



(b) Total finish time

Figure 4.7: P99.9 response time and total finish time of different **idle lengths**

benefits since it does not clean enough data. As a result, BC is still triggered.

The median (P50) response time of AT-IC is larger than that of the no control in all cases. The reason is that with fixed configurations, AT-IC is triggered from the beginning of the workload although there are still plenty available resources in the media cache. For no control, the response time will be only enlarged during a blocking cleaning. The number of requests with enlarged response time is smaller than that of AT-ICs.

Figure 4.5(b) shows workload completion time. The results of $< 20 : 2 >$ indicate that with a too-small idle ratio, AT-IC can not finish the workload earlier. While as the idle ratio increasing ($< 5 : 2 >$), the workload completion time can be reduced by 10% – 15% compared to those without any controls. The reason is that AT-IC avoids the large response time increased by the blocking cleaning. However, if the idle ratio is

too large, e.g., 2 : 2, the total completion time will be enlarged due to the excessive idle duration, even though the blocking cleaning can be avoided.

Idle length. In this experiment, we discuss the performance influences from different idle lengths while keeping the same idle ratio (about 28.75%). Figure 4.6 shows the distribution of the response time. Figure 4.7 displays the P99.9 response time and the workload finish time. Comparing the results of $< 2.5 : 1 >$ and $< 5 : 2 >$, we find that by shortening the idle length, the tail response time of AT-IC can be further reduced. However, if the idle length is too short, AT-IC may have no effects. The distribution of the response time of $< 1.25 : 0.5 >$ is close to that of no control. Since an idle cleaning needs about 250ms to trigger, almost half of the idle time is wasted without cleaning data. The total finish time of $< 1.25 : 0.5 >$ is also larger than those of $< 2.5 : 1 >$ and $< 5 : 2 >$ as shown in Figure 4.7(b).

Workload Characteristics

We have investigated the following factors of a workload including request arrival rates (Arrival Interval), ratios of non-sequential SMR writes (NSSW ratio), counts of non-sequential zones (NSZ) with I/O sizes of 1MB and 8KB respectively. We only show the results of 1MB I/O size in Table 4.2. The evaluations with 8KB I/O size have similar results since the cleaning efficiency of the media cache is determined by the distance between the update offset and the write pointer location [37]. Since we have discussed the detailed results in Subsection 4.3.2, in this evaluation, we demonstrate the tail (P99.9) response time and the completion time of the workload (Comp. Time). In Table 4.2, we synthesize workloads with 35GB total NSSW which is enough to fill up the media cache (28GB). For each workload, we execute the workload with AT-IC ($< Execution : Idle >=< 5 : 2 >$) using independent 16 threads. We compare the P99.9 response time and workload finish time of AT-IC with those of no control (No).

Request Arrival Speed We use the time interval between requests (Arrival Interval) in Table 4.2 to represent the request arrival rate. In this experiment, the non-sequential writes spread over 1024 zones. We set the NSSW ratio to 10% (10% sequential writes and 80% read) and vary the Arrival Interval among 30ms, 50ms, and 100ms. A smaller Arrival Interval means a higher request arrival rate, i.e., the workload is more intense.

Table 4.2: Workload Characteristics (1MB I/O Size)

Arrival Interval				
Value	P99.9 (s)		Comp. Time (s)	
	No	AT-IC	No	AT-IC
20ms	12.99	14.37	7617	8832
50ms	9.22	2.56	24559	22546
100ms	4.32	2.12	40375	38861
NSSW Ratio				
Value	P99.9 (s)		Comp. Time (s)	
	No	AT-IC	No	AT-IC
10%	4.32	2.12	40375	38861
20%	6.78	3.03	22112	19356
40%	12.32	10.22	11018	11128
NSZ Count				
Value	P99.9 (s)		Comp. Time (s)	
	No	AT-IC	No	AT-IC
8	1.1	2.23	36037	38066
1024	4.32	2.12	40375	38861
2048	4.66	2.57	41003	39230

In the results of different Arrival Intervals, both of the tail response time and completion time for no control become larger for the smaller Arrival Intervals. In a more intense workload, e.g., 20ms request arrival interval, the filling speed of media cache is faster than the cleaning speed of AT-IC. Therefore, AT-IC can not avoid blocking cleaning.

AT-IC can achieve better performance with larger Arrival Intervals. When the Arrival Interval becomes 50ms, AT-IC can reduce the tail I/O response time and the total completion time by 72.2% and 8.1% respectively. When the Arrival Interval is 100ms, AT-IC still reduces the P99.9 response time of no control by 47.1%. Therefore, we argue that workload intensity (Arrival Interval) can significantly influence the effect of AT-IC. AT-IC works better in light-weight workloads. With a very intensive workload, AT-IC may need to be reconfigured for a longer idle duration or a higher idle ratio to clean the media cache more efficiently.

NSSW Ratio In the "NSSW Ratio" of Table 4.2, we generate workloads with

different NSSW ratios (10%, 20%, and 40%). We keep the same ratios of SSW (10%) and Read (80%, 70%, and 50%) with 100ms request arrival intervals. NSSW spread among 1024 zones. In the results without control, a higher NSSW ratio results in a worse drive performance since more write requests are blocked during a BC. The service latency of a write is increased more than that of a read based on our previous evaluations (Figure 4.2(b)). The blocked I/O requests accumulate a larger I/O response time with more non-sequential writes.

By comparing the results of no control to that of AT-IC, we find that with 10% and 20% NSSW ratios, AT-IC can reduce the tail response time by 50.9% and 55.3% respectively. The workload completion time of AT-IC is also closer to that of a CMR drive comparing with the completion time without controls. However, with a large NSSW ratio like 40%, AT-IC with the current configuration cannot achieve enough benefits since it can not avoid BCs. Therefore, we confirm that NSSW ratio significantly influences the AT-IC performance. AT-IC may need to be configured with a longer idle duration for workloads with higher NSSW ratios to achieve higher cleaning efficiency.

Counts of Non-sequential Zones In the "NSZ Count" in Table 4.2, we keep the workloads with the same NSSW ratio (10%), and request arrival speed (100ms). However, we spread the non-sequential writes among different numbers of zones (8, 1024, and 2048). When the NSZ count is 8, a blocking cleaning can be finished in a short time. Therefore, the drive does not have exceptionally long I/O response times. When the non-sequential writes spread among more zones, a higher I/O response time is produced because a blocking cleaning will last longer. The drive will suffer a bad performance for a longer duration. I/O Requests thereby accumulate a longer I/O response time.

By comparing the results of AT-IC to those without control, we find that AT-IC is able to reduce the I/O response time when the requests are distributed among more zones. When the count of non-sequential zones is very small, e.g., 8, the P99.9 response time of AT-IC is more than twice of that without controls since the response time is enlarged by the induced idle duration. Therefore, for workloads with a small count of non-sequential zones, AT-IC may not be helpful. A blocking cleaning can finish in a very short time without influencing the drive performance significantly.

Conclusions We have validated that AT-IC can significantly reduce the tail response time of HA-SMR drives. Both idle configuration and workload characteristics significantly influence the effects of AT-IC. Carefully configuring the idle length and idle ratio is required if AT-IC wants to achieve excellent performance benefits. However, the characteristics in a real workload will dynamically change from time to time. They are harder to handle by AT-IC with a fixed idle configuration. Besides, as we discussed, AT-IC with fixed configurations has a long median (P50) response time, indicating AT-IC blocks more requests than no controls. The reason is that AT-IC does not consider the current utilization of the media cache and starts idle duration from the beginning of the workloads. Therefore, in the next section, we will discuss how to improve the drive performance by using artificially triggered idle cleanings with adaptive idle configurations considering both the utilization of the media cache and workload characteristics.

4.4 Idler: Artificially Triggering Cleanings with Adaptive Idle Durations

4.4.1 Basic Principle

Idler can trigger idle cleaning using adaptive idle durations based on the current media cache resource utilization as well as real-time workload characteristics. Typically, Idler needs to decide when to trigger an idle duration and how long the idle duration should be. Based on the previous evaluations in Section 4.3.2, Idler considers the following parameters: $\boldsymbol{U}$ is the media cache resource utilization including free space and mapping slots. We set two thresholds, U_{low} and U_{high} ($0 < U_{low} < U_{high} < 100\%$). $\boldsymbol{T}_r$ is the average time interval between consecutive requests. $\boldsymbol{T}_{lat}$ is the average request latency. $\boldsymbol{V}_m$ and $\boldsymbol{V}_c$ represent the consuming and cleaning speeds of media cache resources respectively. $\boldsymbol{L}$ is the length of the duration to trigger idle cleaning.

These parameters are continuously monitored, and the conditions are periodically evaluated. Both media cache space utilization and the usage of mapping entry slots are monitored and evaluated with two thresholds U_{low} and U_{high}. Since blocking cleanings can be triggered either by depleted space or mapping entry slots, U will consider the resource, space or mapping entry slots, closer to be depleted to represent the media

cache utilization. When $U < U_{low}$, Idler will not trigger an idle duration since a good amount of resources are still available in media cache. When $U_{low} < U < U_{high}$, we start to trigger an idle duration. The idle duration length L is decided based on Equation 4.1. Since media cache still has a certain amount of resources, Idler will not trigger a long idle duration to avoid introducing a large I/O response time. We calculate the average latency of the requests T_{lat} and compare T_{lat} with the average time interval between requests T_r. If $T_r \leq T_{lat}$, it means the requests are issued relatively intensively. In this case, Idler will not trigger an idle duration ($L = 0$) since any idle duration may block a large number of requests. When $T_r > T_{lat}$, there are some time intervals between requests. In this condition, Idler triggers the idle duration with a minimal length to limit the maximal I/O response time. We set the minimum length to 0.3s here to make sure that idle cleanings will be triggered.

$$L = \begin{cases} 0, & T_r \leq T_{lat} \\ 0.3s, & T_r > T_{lat} \end{cases} \tag{4.1}$$

When $U > U_{high}$, Idler has to clean the media cache more aggressively. In this condition, L is determined by Equation 4.2. If the workload is not intensive ($T_r > T_{lat}$), Idler will trigger an idle duration with a maximal length to achieve a higher cleaning efficiency. We set the maximal length to 2s which is equal to the largest service latency when a blocking cleaning is triggered to limit the maximal I/O response time. If $T_r \leq T_{lat}$, it means the current workload is intensive. A large number of requests will be delayed during an idle duration. To clean the media cache more efficiently and minimize the I/O response time, Idler will further compare the consuming speed V_m to the cleaning speed V_c of the media cache resource. If $V_m > V_c$, it means U will keep increasing. Idler has to increase L to achieve better cleaning efficiency. When $V_m \geq V_c$, Idler can decrease U with the current L. Therefore, Idler reduce the L to reduce the I/O response time further.

$$L = \begin{cases} L - 0.1s, & T_r \leq T_{lat} \ \& \ V_m < V_c \\ L + 0.1s, & T_r \leq T_{lat} \ \& \ V_m > V_c \\ 2s, & T_r > T_{lat} \end{cases} \tag{4.2}$$

4.4.2 Idler Implementation

Media Cache Resource Monitoring Monitoring media cache resources is challenging. There are no available interfaces to check the media cache resource utilization directly. The only useful interface is the *report_zone* interface in libzbc [36, 111]. It can check the states of zones (sequential or non-sequential) and the write pointer location of each zone. The *report_zone* is an intrusive command incurring a relatively large performance overhead. Therefore, it cannot be called frequently. However, we do not need the precise values of these parameters.

To reduce the performance overhead, we only estimate the media cache resource utilization U instead of getting an accurate value. Idler maintains a ghost media cache which records the total amount of consumed resources (both space and mapping entries). It also remembers the media cache resources used by each zone and the write pointer location of each zone. For non-sequential writes directed to media cache, Idler keeps updating U. For sequential writes, Idler will also update the location of write pointers. Since data is buffered in media cache with a log structure [109], repetitive updates to the same address will also be counted.

We periodically update the information in the ghost media cache by checking the current zone states using *report_zone* (every 10 minutes). By comparing the set of non-sequential zones in ghost media cache with that from the drive media cache, we identify the cleaned zones and the cleaning speed (V_c) during the last period. U can be updated by eliminating the resources used by the cleaned zones.

Workload Characterization Idler calculates request intervals T_r, the average latency of requests T_{lat} and the consuming speed of the media cache resource V_m during the execution duration. After an idle duration, both of the blocked requests and newly issued requests will be served during the execution duration. Since the blocked requests are issued in an intensive way without following their original time intervals, the calculations of T_r and V_m are only performed after we finish all of the delayed requests. For the un-delayed requests, Idler samples 10 consecutive requests to calculate T_r, T_{lat} and V_m. T_r and T_{lat} will be counted for each new request, while V_m will be counted for only non-sequential writes.

Idle Frequency The idle frequency is determined by both idle duration and execution duration. In Idler, the execution duration is not fixed. The length of an execution

duration is long enough to serve all blocked requests from the previous idle duration plus some number of new requests such that enough new requests can be sampled to determine the next idle duration. If Idler decides not to trigger an idle duration, a new execution duration will start right away.

4.4.3 Integrating Idler with A Write Buffer

In many scenarios, storage systems may include a high-speed write buffer to speed up the writes. The difference between an external write buffer and the on-device media cache is that writing to the external buffer will not interrupt the data cleaning on SMR drives. Although writes to and reads from an write buffer may create idle durations, the triggered idle cleaning may not clean enough data in media cache since uncontrolled buffer evictions and buffer-missed reads will interrupt the idle cleanings. Idler can also be helpful when it is integrated with a write buffer to control the page evictions and buffer-missed reads. Besides, this write buffer also provides opportunities to reduce the I/O response time introduced by the triggered idle durations since it can temporarily hold writing requests during the idle duration. The data temporarily buffered at the write buffer will eventually migrate to SMR drives.

Designing an efficient write buffer for SMR drives is not the primary contribution of this paper. Existing studies have discussed multiple designs to write buffers for SMR drives [37, 112, 113, 114]. Our goal here is to study how Idler designed for SMR drives can work together with a write buffer in SSD to further improve the performance of SMR drives. To simplify our discussion, we make the following assumptions for a write buffer.

The write buffer is a page-based write buffer on SSD. Here we set the page size to 4KB. A read request will check the buffer first and then go to SMR drives if they cannot be satisfied from the buffer. We assume a simple eviction policy – the Zone-based Least Recently Used (ZLRU). ZLRU remembers the access time for all of the zones who have data buffered. The victim zone will be selected by LRU. All of the buffered data from the same zone will be evicted together. We have two data paths when evicting data from the write buffer to SMR drives. If we have more than 400 pages (i.e., 1.6 MB) to be cleaned for a zone, we write the data back to its designated zone directly by read-merge-writes. Otherwise, we evict the data to the media cache. We choose 400 pages as

a threshold since we assume the average service latency for a 4KB write is about 10ms, and it takes about 4 – 6s to read a zone out, reset the write pointer and write data back.

The write buffer consists of a free page pool and a page map. We load the map into memory and persist it in a reserved region on SSD. The write buffer allocates new pages from the free page pool to buffer new writes. For SMR drives, a victim zone list is maintained based on ZLRU. If not enough free pages are available, page evictions will be triggered. The reclaimed pages will be returned to the free page pool.

We integrate Idler with a write buffer in the following way. During an idle duration, the write buffer will temporarily hold both non-sequential writes and sequential writes if it has free space. Reads will not be delayed if they can be read from the buffer. At the meantime, we set up a wait queue to receive requests during an idle duration under two conditions. First, if the buffer is full and the data to be written are not in the buffer, the writes will be put into the wait queue. Second, reads will also be put into the queue if they cannot be satisfied from the buffer. In the next execution duration, we will issue the delayed reads and SSWs in the wait queue to SMR drives. The delayed NSSWs will be buffered in the write buffer after zone evictions.

The basic principle discussed in Section 4.4.1 is still applied. That is, both workload characteristics and media cache resource utilization will be considered when deciding the length of an idle duration. We still use the ghost media cache, as discussed in Section 4.4.2, to estimate the media cache resource utilization. However, instead of increasing media cache resource utilization for each non-sequential writes, we update the used media cache resource information during the eviction. If the data is evicted, we increase the media cache resource utilization accordingly in the ghost media cache. We still periodically (every 10 minutes) check the zone states by *report_zone* interface and adjust media cache resource utilization(U) and the cleaning speed (V_c).

We characterize the workload during the execution duration. The average request interval and the average request latency are counted only for the buffer-missed requests (including non-sequential writes, sequential writes, and reads) since buffer-missed requests will probably be delayed during an idle duration. When calculating the consuming speed of the media cache resource V_m, we only count the non-sequential writes that do not have a hit in the buffer. The non-sequential writes will not increase the media

cache utilization if they can be served from the buffer.

4.5 Idler Evaluation

We implement Idler as a block I/O replay engine and evaluate it by executing two real-world traces [115]. To emulate the different I/O dependencies, we still partition the traces into different numbers of sub-traces and replay with separate threads. Again the requests in the same partitions are dependent. Precisely finding I/O dependencies from block traces is hard without extra information. HFReplayer [116] can approximately estimate I/O dependencies in block traces based on the latency information. Therefore, we use a similar algorithm when partitioning a trace.

When executing the traces, we respect the request intervals between the requests to emulate their characteristics. We run the traces on an HA-SMR drive using No Control (regular operation without idle durations), AT-IC, and Idler respectively. After we try multiple times, we configure AT-IC with $< 5 : 2 >$ since it provides a better result. For Idler, we set $< U_{low}, U_{high} >$ to $< 0.1, 0.8 >$ to allow the media cache to buffer enough data before an idle cleanings is triggered. And after U_{high} is reached, the media cache still has available resources. We also execute the traces on a Conventional Magnetic Recording (CMR) drive and compare the workload completion time.

The characteristics of the workloads are shown in Table 4.3 including the counts of writes, the total write sizes, counts of non-sequential zones, and the write ratio. In these traces, blocking cleanings are triggered by depleting the mapping entry slots, although the writing sizes of traces are not significantly larger than the size of media cache. All traces are captured on a CMR drive. Almost all of the writes are non-sequential. During a workload, requests are not issued with a unified rate. Figure 4.8 uses the trace proj_1 as an example to show the variation of I/O counts for every 100 seconds within an hour. The intensity of the workload varies a lot with time. We partition the trace into 1, 16 and 128 sub-traces to emulate different conditions of I/O dependencies.

Table 4.3: Characteristics of traces (proj_1, prn_1)

trace	write counts & sizes	zones	write%
proj_1	2,496,935 & 25.576GB	1,475	10.56%
prn_1	2,769,610 & 30.785GB	1,222	24.66%

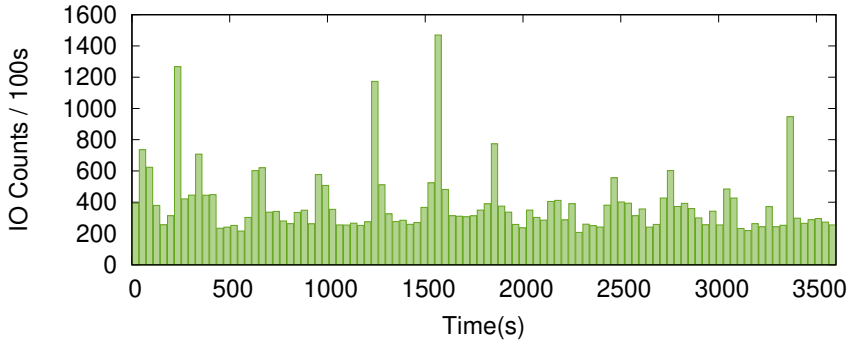


Figure 4.8: The variation of the request intensity in proj_1

Figure 4.9 shows the response time of three designs in non-cleaning condition (the first 1,000s) of the workload under 128 I/O threads. During the first 1,000s, No Control has the shortest response time since there are no artificially triggered idle durations delaying the subsequent requests. However, with a fixed configuration, the response time of AT-IC varies a lot. When the workload is I/O intensive, the predefined execution time may not be able to finish all of the blocked requests from the previous idle duration. Thus, AT-IC accumulates a long response time. The proposed Idler can adjust its idle length and execution length dynamically. When the workload is intensive, Idler will keep the idle duration short if the media cache still has a lot of available resources. Even with a longer idle duration, Idler can make sure that the next execution duration will finish all of the blocked requests from the previous idle duration. Therefore, Idler can keep the response time in a small value (mostly under a second) even though they are still longer than those of No Control when no blocking cleaning is triggered.

We also compare the P99.9 response time and workload finish time in Figure 4.10. The results of no control indicates that the blocking cleaning is triggered although the workload may have some burstiness. The idle cleanings triggered by the natural idle duration cannot clean the media cache efficiently. Idler is able to provide better response time than AT-IC and no control since Idler only triggers minimal required idle durations to avoid blocking cleanings with limited performance overheads. Under a single thread, Idler reduces the P99.9 response time and workload finish time by 56% and 23% compared with those of without a control. The improvement is more significant if having more independent I/O requests for Idler. The P99.9 response time is reduced

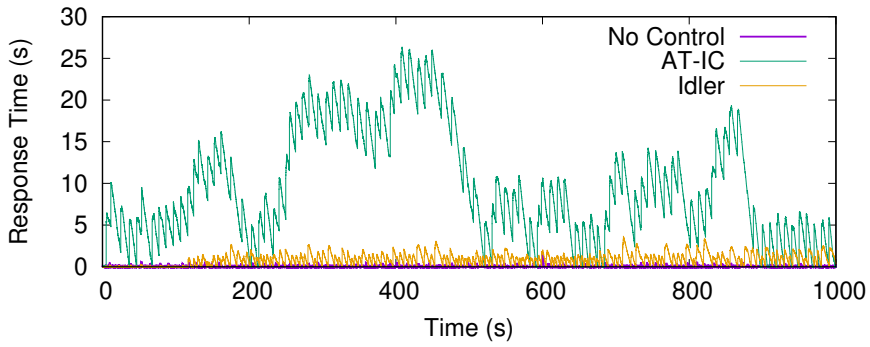
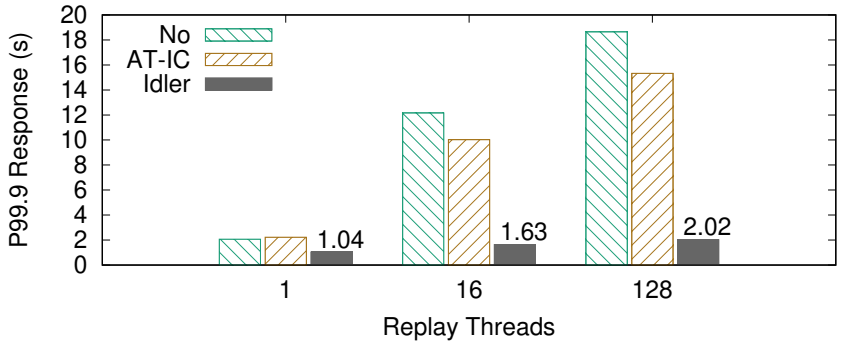


Figure 4.9: Response time of proj_1 **before BC starts** with 128 threads

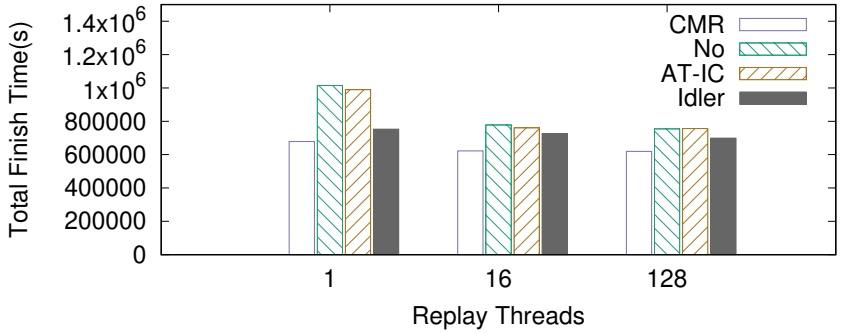
by 83.2% and 88.8% with Idler compared to the results without control under 16 and 128 threads respectively. However, the tail response time of Idler is still not comparable with that of a CMR drive, even though the total completing time of the workload with Idler is closer, about 8% – 10% longer, to that of a CMR drive. Results in Figure 4.10(b) indicate Idler can finish the workload in a shorter time. Since we issue the requests according to their timestamp when replay the trace and no blocking cleaning is triggered, the total finish time of Idler under different numbers of I/O threads is close to each other.

Idler needs extra memory and computing resources for buffering the delayed requests during an idle duration and updating the ghost cache information. Therefore, we measure the memory and CPU overheads of no control, AT-IC, and Idler while replaying the trace proj_1 with different numbers of threads. The maximum memory consumption is similar under different I/O threads which are about 3.9MB (no control), 4.2MB (AT-IC) and 5.1MB (Idler). Idler does not introduce a significant memory overhead since it keeps the idle duration small and will not buffer too many requests. However, Idler needs some memory space to store the ghost cache information. Idler does not introduce a significant CPU overhead either since updating ghost cache information for each request only modifies several variables and *report_zone* is called infrequently. The differences in CPU utilization among no control, AT-IC, and Idler under different I/O threads are less than 10%.

However, the results of Figure 4.11 shows that when the workload has a higher ratio of non-sequential writes (24.66%), both AT-IC and Idler can not help to reduce the



(a) P99.9 response time

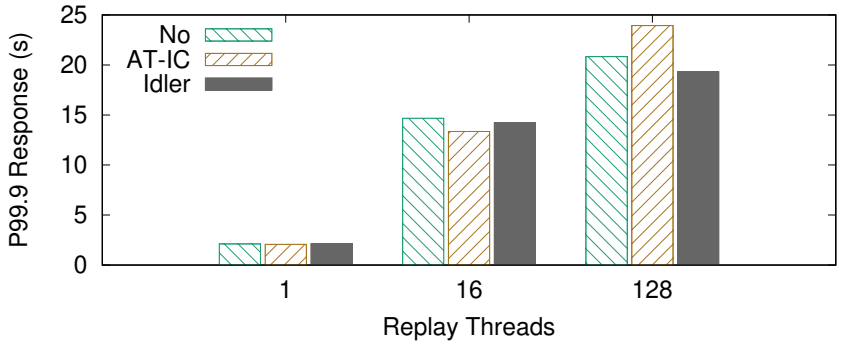


(b) Total finish time

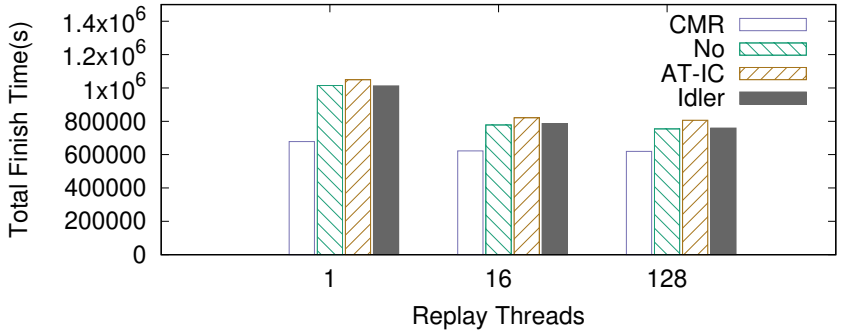
Figure 4.10: P99.9 response time and total finish time of **proj_1**

response time. The response time of AT-IC and Idler are similar to those of no controls. Both of the P99.9 response time and workload completion time are close to those of no controls. Therefore, we recommend that Idler can be deployed in the workloads with low ratios of non-sequential writes under 10%.

We further evaluate Idler with more traces with 16 I/O threads. Those traces include enough non-sequential writes and non-sequential zones so that the blocking cleaning will be triggered if the drive has no control. The non-sequential write ratios and the non-sequential zone counts of these traces are (8.53%, 1886) for usr_1, (18.85%, 926) for usr_2, and (12.39%, 1474) for proj_2. Figure 4.12 shows the tail response time of these three traces. The results indicate that Idler can significantly reduce the tail response time in the workloads with a small non-sequential write ratio (under 10%) compared with the results with no control.



(a) P99.9 response time



(b) Total finish time

Figure 4.11: P99.9 response time and total finish time of **prn_1**

Idler can reduce the response time compared to that without any control in workloads with low non-sequential write ratios. However, the response time of Idler can still be large. The P99.9 response time can be up to seconds depending on the configurations of maximal idle length. Besides, for workloads with high non-sequential write ratios, Idler can not avoid the blocking cleaning. Therefore, we further integrate the Idler with a write buffer on SSD. The buffer can help to absorb the non-sequential writes and temporarily hold the requests during an idle duration so that the response time of Idler will be further reduced. To demonstrate the effectiveness, we only use a small buffer (64MB), although better results can be achieved with a larger buffer. When executing workloads on the CMR drive with a write buffer, we use regular LRU instead of ZLRU evictions.

Figure 4.13 and Figure 4.14 shows the P99.9 response time and the total finish time

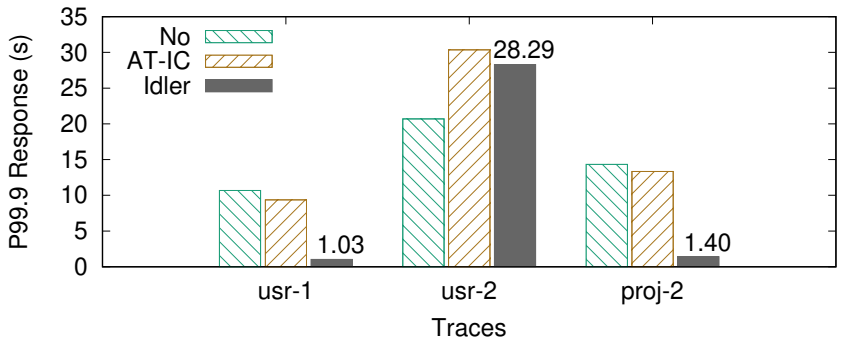


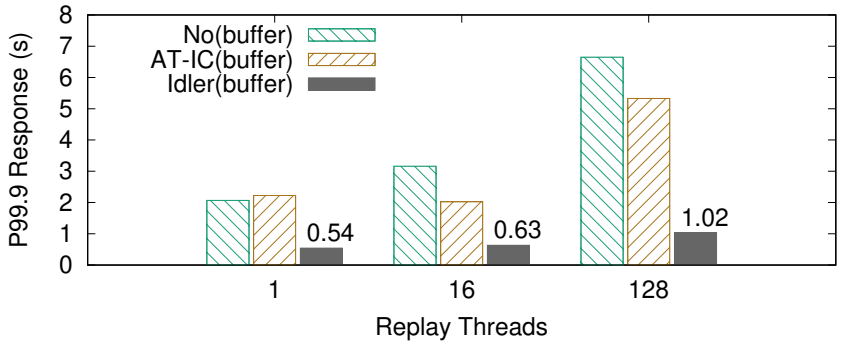
Figure 4.12: P99.9 response time of different traces

of Idler with a write buffer. For "No" and AT-IC, a write buffer cannot reduce the tail I/O response time since blocking cleaning is triggered. The page evictions and buffer-missed reads are still delayed. The results of Idler indicate that with the help of a small write buffer, the P99.9 response time can be further reduced compared to those without a write buffer in Figure 4.10(a). When the ratio of non-sequential writes is small (Figure 4.13), the P99.9 response time can be shorter than a second. In the workloads with 10% ratio of non-sequential writes, the P99.9 response time is reduced by 71.3% – 87.7%. Meanwhile, Figure 4.14 shows that with a write buffer, Idler can reduce the P99.9 response time by 70% – 85.2% in a workload with a higher ratio of non-sequential writes. The reason is that the write buffer absorbs a good amount of non-sequential traffics, Idler only triggers short idle durations.

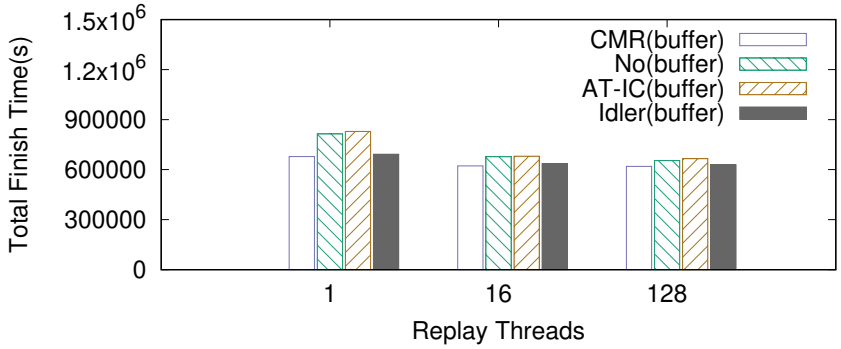
4.6 Related Work

Most of the existing researches including file systems [117, 118], key-value stores [119, 120] and other storage systems [121, 122, 123, 124] target at HM-SMR drives. However, applications or systems have to be modified significantly or even re-designed since HM-SMR drives do not accept non-sequential workloads. There are also other studies which intend to help HM-SMR drive handle non-sequential workloads [125]. The concept is similar to the STL in DM/HA-SMR drives.

For DM/HA-SMR drives, existing studies focus on performance modeling and evaluations of the drives. Skylight [109] and Wu et al. [37] conduct evaluations on DM-



(a) P99.9 response time

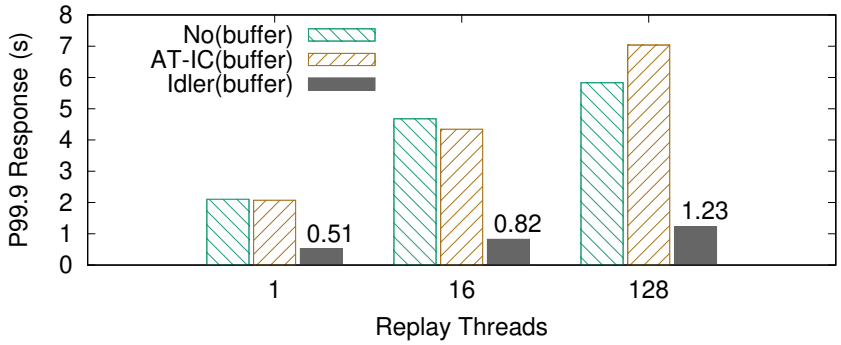


(b) Total finish time

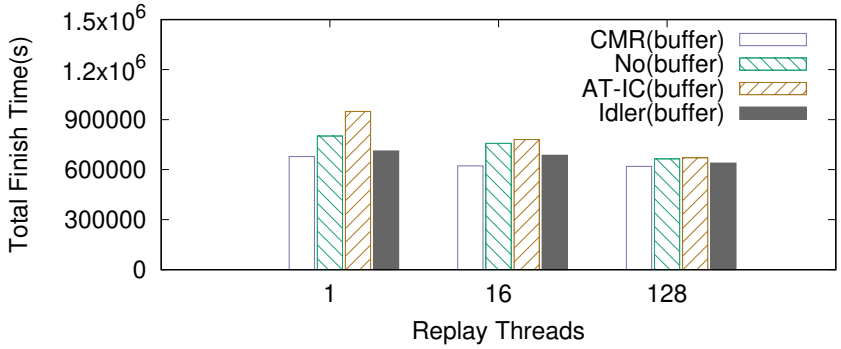
Figure 4.13: P99.9 response time and total finish time of **proj_1** with a buffer

and HA-SMR drives respectively to find the internal structure, cleaning policies, etc. Mhalagi develops a simulator to simulate the drive performance [126]. Shafaei also accurately models the performance of DM-SMR drives [127]. Aghayev involves ext4 in DM-SMR drives [128, 129].

Besides, there are other designs for HA-SMR drives to avoid the long response time. H-Buffer [37] and VPC [102] intend to improve the performance of HA-SMR drives by utilizing a log-structured buffer/cache on shingled zones in additions to the Media Cache. SMRC is another design using an SSD as the upper-layer cache for SMR drives [114]. Idler can also be integrated with them.



(a) P99.9 response time



(b) Total finish time

Figure 4.14: P99.9 response time and total finish time of **prn_1** with a buffer

4.7 Conclusions and Future Work

In this paper, we evaluate the cleaning process of HA-SMR drives. We identify how data cleaning affects the drive performance. We propose a scheme called Idler. Our evaluations show that Idler can avoid the extremely long and unpredictable I/O response time in workloads with an NSSW ratio smaller than 10%. In the future, we would like to deploy Idler with some applications like database and key-value store to verify the performance improvement.

Chapter 5

Improving Data Integrity in Linux Software RAID using Protection Information

5.1 Introduction

Silent Data Corruption (SDC) often causes data loss that can go completely undetected without any notifications, warnings, or error messages. One example of SDC is bit-flips caused in memory media by cosmic rays or by-products of particle decay [12]. DRAM errors occur at 2% rate of all Dual Inline Memory Modules (DIMM) each year [130]. SDC errors are often not picked up by Error-Correcting Code (ECC) mechanisms that are employed in enterprise-class DIMMs either [131]. The DIF (Data Integrity Field) and DIX (Data Integrity Extensions) [132] proposed by the T10 Technical Committee provide a mechanism to address SDC in storage systems. DIF (also known as T10 Protection Information), adds an additional 8 bytes of integrity data to each 512-byte data sector on disk. The first 16-bit guard (GRD) tag is the CRC or IP checksum of the 512-data portion followed by an user-defined 16-bit application (APP) tag. The last 32 bits are the reference (REF) tag, which typically contains the LBA offset, to detect out-of-order or misaligned writes. Typically, PI is generated during a write and verified during a read by the HBA in order to ensure the data integrity from HBAs to devices

[16].

DIX extends DIF by enabling applications and operating systems to generate Protection Information (PI) along with I/O requests, and verifying it at each layer of the I/O stack from the source of the request to the disk. Figure 5.1 demonstrates the key layers in I/O stack protected by the DIF and DIX. Since operating systems deal with data in page sizes, the scatter-gather list structure is utilized to store the data and PI in memory and later merged in the HBA [38].

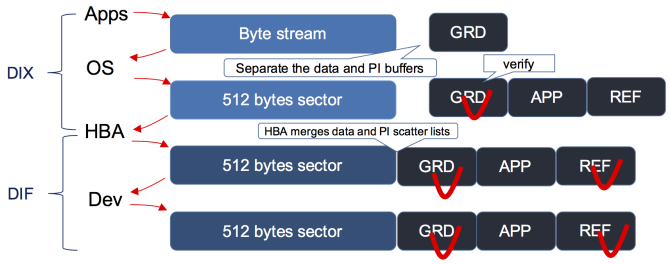


Figure 5.1: I/O stack protected by DIF/DIX

Redundant Array of Independent Disks (RAID)[133] is predominantly used in high-performance storage arrays to provide reliability of data by either block-level data replication or data parity computations to recover from data loss. Software-based RAID [134] has gained mass adoption, thanks to the MD module in the upstream Linux kernel. It offers functionality that is similar to hardware RAID controllers at a fraction of the cost and management overheads. The current RAID-5/6 implementations, which are the de-facto RAID levels used in commercial storage arrays, do not take advantage of DIF and DIX capabilities of PI-capable drives nor do they make sense of the PI embedded within I/O requests coming from upper layers in the stack. As a result, although applications or the OS may generate PI for data and include them with I/O requests, only the data portion of the request is processed by MD, and the integrity metadata is eventually lost. Moreover, the MD does not generate PI for the P and Q parities that are generated as part of the RAID 5/6 layouts either.

To address this challenge, we propose a DIX-aware MD module design that improves data integrity in MD. Specifically, the MD driver's RAID-4/5/6 personality module has been enhanced to take advantage of DIX to generate and verify data integrity protection information and close the integrity gap that was otherwise present in such systems.

5.2 Related Work

The developments of modern storage systems have two main trends. First, the capacity of storage systems are becoming larger and larger. In this aspect, some researchers tried to deploy new devices with larger capacities. Wu et al. exploring the solution for Shingled Magnetic Recording Drives [37]. Another trend is to increase the performance of systems. Non-Volatile Memory is studies as Cache for data to gain better performance [135, 136].

There have been numerous studies that focus on the detection and correction of SDC in storage systems. Berrocal et al.[12] presents a point-wise detection model to detect corruption indirectly with limited monitored data size. With extra memory overhead and computing time, a large number of corruption can be detected after a two-phase detection. nZDC[11] is an effective instruction duplication based approach to against SDC. The micro-architectural components could be protected without any performance penalty. Di et al. [137] propose multiple linear prediction methods to improve the SDC detection results. The detection sensitivity is improved by an error feedback control model and the overhead is minimized by the spatial-data even-sampling method. Another novel technique to detect silent data corruption by data monitoring is proposed by Baultista et al. [138]. Applications could quickly spot anomalies by learning from normal dynamics of its datasets.

The Linux kernel I/O stack already supports T10-PI to allow for SDC detection and correction[132]. Embedding data checksum along with the on-disk data sectors allows detection of corruption at any layer in the data path. In the software RAID layer, the RAID-0 and RAID-1 personalities already support T10-PI[139]. However, in RAID 0/1, the PI is merely passed along without verification. T10-PI is also supported by several commercial RAID vendors. Some vendors have provided T10-PI support in hardware, e.g. Dell PowerEdge RAID Controller[140]. In such solution, the PI gets generated and verified within its hardware RAID controller and then passed to lower I/O layers.

Beyond the block layer, several filesystems also support end-to-end integrity using checksums. BTRFS[141] is a local filesystem that can verify data checksum and detect corruption and misaligned writes. Lustre, a widely used distributed filesystem in HPC, also has a future roadmap feature for supporting end-to-end data integrity from the

clients using T10-PI. A design[142] has already been proposed and implemented for such a feature. This design, when used in conjunction with our DIX-aware MD module, can provide *true* end-to-end data integrity guarantees on supercomputing systems.

5.3 Design and Implementation

The current Linux I/O stack uses the Scatter-Gather list structure in the kernel to implement DIX. I/O requests are constructed independently for data and PI. As mentioned in Section 5.1, current software RAID implementation in the MD module does not verify data integrity to leverage the PI passed along with the I/O request. Figure 5.2 demonstrates a write example in a 4-device raid 5. The basic unit of a write request is the stripe head structure, which consists of single page-sized device buffers for corresponding offsets on each device. The RAID-5/6 personality of the MD module splits write requests by page size and copies the data from page0-page2 in the request to data buffers D0-D1 in the stripe head structure. This is done to compute the parities according to the request's target address. New per-device write requests are compiled with these data buffers and submitted to the lower levels in the block I/O stack. Splitting out the larger request into per-device requests increases throughput by processing them in parallel across devices. Depending on the RAID layout, this can also boost reliability. However, when the MD request is broken down into per-device requests, PI is lost. The same issue exists with read requests. When both data and PI are read from devices, only data is retained in the stripe head structure's data buffers. This loss of integrity data in the MD module creates an integrity gap in the Linux I/O stack from MD module to I/O controller, which is marked in Figure 5.2.

To close this integrity gap, we propose a DIX-aware design of MD module. In this design, a PI operator, which is able to finish general PI operations, is added to the MD module. There are four main types of PI operations: *no-operation, preparation, verification, and generation.* A no-operation indicates there is nothing to do for the request. A preparation prepares the request for PI, including checking if the target device is PI-capable and allocating a PI page for the request if needed. A verification verifies that data match its PI. A generation generates the PI and stores it in the PI page allocated by the preparation operation.

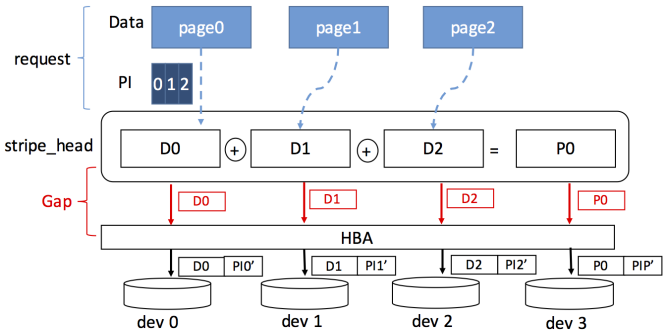


Figure 5.2: The integrity losing problem

In addition to the PI operator, dedicated PI buffers, which are demonstrated in Figure 5.3, are allocated and managed in the stripe head structure to store and operate the PI. With these dedicated PI buffers, the PI is stored and operated (verified or generated) in the MD module and finally passed to the lower block level together with data. In this way, the integrity gap in a generic Linux I/O stack that was caused by losing integrity data in the MD module is filled in.

The DIX-aware MD module has different processes when dealing with write and read requests.

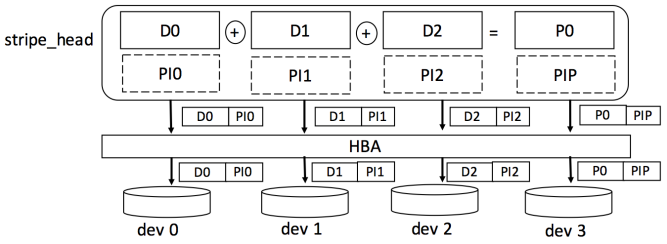
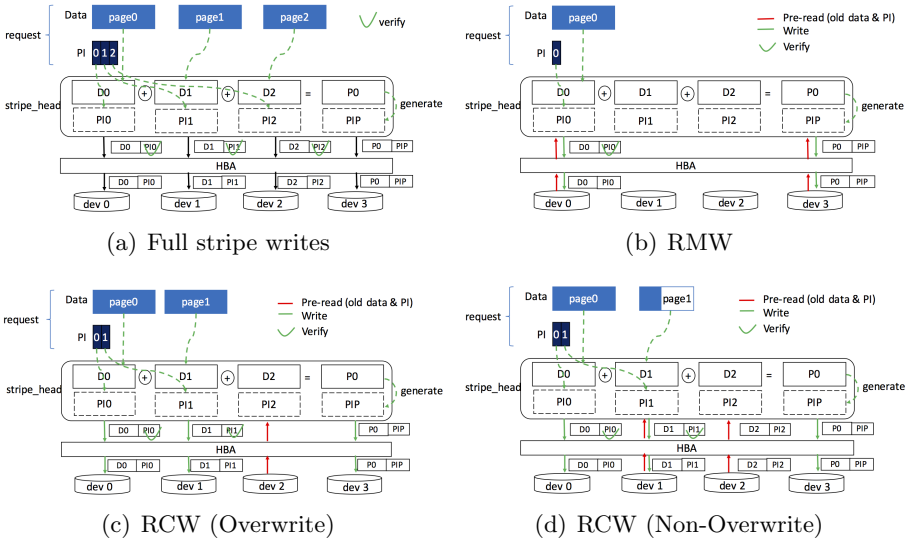


Figure 5.3: DIX-aware MD stripe head structure

5.3.1 Read requests

Figure 5.5 describes a stripe read process. For large read requests that are spread across multiple devices, both data and PI are read by small requests made directly to devices and then stored in data buffers D0-D2 and PI buffers PI0-PI2 respectively. After the reads requests complete, the data and its corresponding PI are copied to the



(a) Full stripe writes (b) RMW

(c) RCW (Overwrite) (d) RCW (Non-Overwrite)

Figure 5.4: The write request processes

data pages and PI pages to fulfill the large read request. After copying the data, a verification for PI and data is performed on the large request to ensure there are no data errors. The verification is performed at this point rather than right after reading from devices because the read process can be considered complete only after copying the data to the large request. There is no need to verify the PI in temporary buffers and any mismatches here are be captured in later verification. After verifying the PI, if no mismatch is found, the large request is returned to the upper levels. Otherwise, a mismatch error is captured and the entire read request is redone.

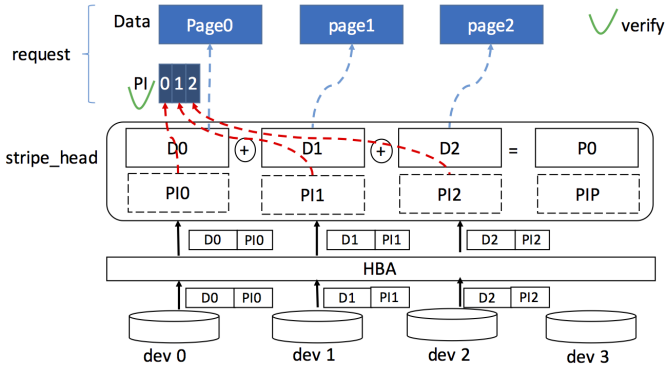


Figure 5.5: The stripe read process

For small read requests that can be handled by a single device, there is no need to use the stripe head structure because the read operations do not compute parities. For these chunk-aligned reads, the PI operator prepares the request by allocating PI pages to the request, after which the well-prepared read request is submitted to the lower block level. Both data and PI are read and stored in the data pages and PI pages in the request. After the read operation, a verification is performed on the request.

5.3.2 Write requests

When a write request comes into the MD module, the PI operator generates PI if it does not exist or performs verification otherwise. With PI buffers in stripe head structure, the PI will be copied along with the data and stored in PI buffers. Figure 5.4(a) demonstrates a full stripe write, in which the data and PI from the requests will fulfill the stripe head structure's data buffers and PI buffers. The parities can be computed by the data from the request directly. If it is a partial stripe write, there are two ways to calculate parities in RAID-5, namely Read-Construction-Write (RCW) and Read-Modify-Write (RMW). Figure 5.4(b) displays a RMW example. The old data and the old parities are pre-read together with its PI from device 0 and device 3 and stored in buffer pairs {D0, PI0} and {P0, PIP}. Then data in page 0 and its PI from request will be copied to buffer pair {D0, PI0} after new parities are computed and stored in P0. Figure 5.4(c) presents a general process of Overwrite RCW, in which all of the data pages in the write request are filled up. The buffer pairs {D0, PI0} and {D1, PI1} store the data and PI copied from requests data and PI pages. Then the data and PI on device 2 are read and stored in {D2, PI2} to compute the parities. Figure 5.4(d) shows the steps of Non-Overwrite RCW, in which one of the page, page1, in request is not filled up. In this case, besides data and PI on device 2 are read and stored in {D2, PI2}, the data and PI on device 1 corresponding to page1 will also be read and stored in {D1, PI1}. Finally, page0-page1 and its PI in the request will be copied to {D0, PI0} and {D1, PI1} respectively to compute the new parities.

After computing the parities, the PI operator generates PI for the parities. Then, small write requests issued directly to the device are built by the data buffer/PI buffer pair. At this point, the PI operator verifies the newly generated write requests to ensure the correct data is being submitted to the lower block level. The verification is

performed at this point because of the complex calculations performed when copying from requests. Also, the next verification will be done on the I/O controller after the data has passed through I/O scheduler, block layer, and SCSI layers, which is far away from the MD module. Thus, any mismatch should be captured as early as possible so that the entire write request can be retried.

The DIX-aware MD module allows PI generated, verified and passed by using a PI operator and improved stripe head structure with dedicated PI buffers. The end-to-end data integrity in the Linux I/O stack is enhanced by closing the integrity gap created when MD requests are spread across multiple devices.

5.4 Performance Evaluation

We carried out a comprehensive evaluation on our DIX-aware MD module, in a testing environment that used actual PI-capable devices, rather than emulating with a SCSI_debug module. Ten PI-capable drives were connected to a host server in Just a Bunch of Disks (JBOD) and formatted in protection Type 1. We also used the IP checksum, which is the more widely used checksum type in the current DIX, as the guard tag. With these 10 drives, we built a RAID-6 (8+2) with the MD module (kernel version v4.1), and conducted several experiments contrasting this module with an unmodified MD module. This enabled us to identify the influence of PI-related operations on storage performance. we used an fio benchmark operating on the raw MD device with direct I/O and disabled all of the on-device write buffers and read caches at the same time. Generally, we considered four I/O types–sequential reads/writes and random reads/writes. The detailed environment configurations are shown in Table 5.1.

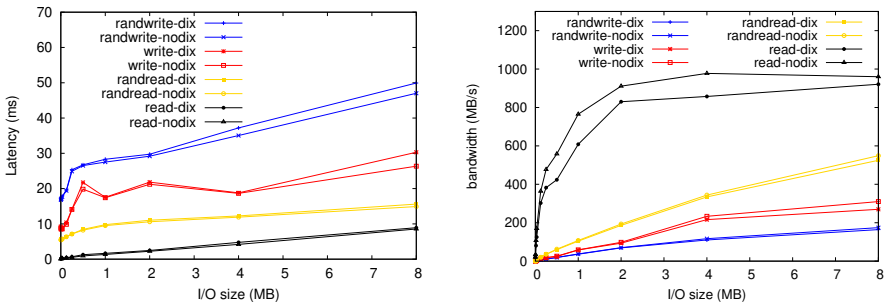
Table 5.1: Experiment environment configurations

Items	Info
CPU	24 * Intel(R) Xeon(R) CPU @ 2.40GHz
HBA	LSI SAS3008 SAS-3 (rev 02)
Driver	mpt3sas out-of-box version 13.00.00.00
Drives	10 * SEAGATE ST6000NM0034

The first experiment was to verify the effectiveness of the DIX-aware MD module in detecting SDC, which was the primary motivation behind designing the DIX-aware MD module. We temporarily added to the MD module a Corrupted Data Generator (CDG), in which the data or the LBA in the request would be modified to incur guard tag or reference tag mismatching. In this experiment, after MD built the new small request to device, the CDG was called to generate a guard tag and a reference tag mismatches in a 2 GB write request with 4 KB I/O size. After observing the kernel log, we found that both guard tag and reference tag mismatches could be detected in our DIX-aware MD module with error messages indicating locations and types of errors. The sample error messages are shown as bellow.

sdg: ref tag error at location 310839 (rcvd 134528567)

sdi: guard tag error at sector 766864 (rcvd c231, want c2ad)

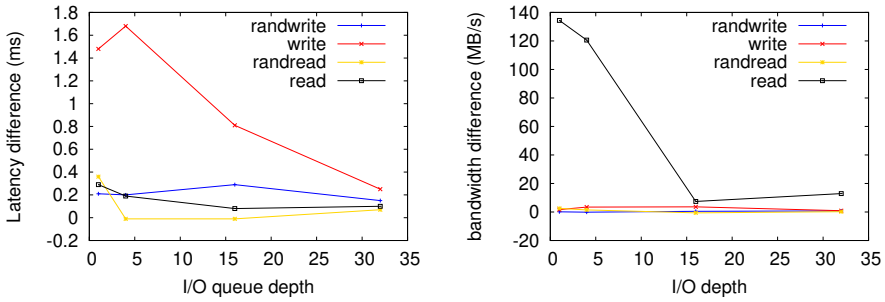


(a) Latency differences (b) Bandwidth differences

Figure 5.6: I/O queue depth influence

After verifying the effectiveness, we studied the influence on I/O latencies from PI operations. The total latency mainly includes the I/O processing time, PI operating time, disk seeking time, I/O wait time and etc. After setting the I/O queue depth to 1 and the RAID chunk size to 512KB, we executed same I/O traces, which consisted of 2KB, 4KB, 16KB, 32KB, 128KB, 256KB, 512KB, 1024KB, 2048KB, 4096KB and 8192KB I/O request sizes for a total data size of 8GB, on both the DIX-aware MD module and the unmodified MD module. During the executions, the I/O latencies were recorded. The results are shown in Figure 5.6(a).

After summarizing the results, we found that in each I/O type the latency of the DIX-aware MD device was larger than that of the original MD device and the latency



(a) Latency differences (b) Bandwidth differences

Figure 5.7: I/O queue depth influence

difference grew as the I/O size increased. We then used a growing value, which was the values of latency increase in the DIX-aware MD module, and a growing ratio, which was the percentage occupied by the growing value, to evaluate the influence on latencies. Even though random requests have larger latency growing values, the latency growing ratios are relatively small (4%-15%) compared to sequential requests (15%-20%). Because the latencies are dominated by the seeking time in a random request, the latency caused by PI operations only took up a small fractions of total latency. In sequential requests, we observed that the latency growing values in write requests (0.02ms-5ms) were much larger than those in reads (0.02ms-0.7ms). This is because there are more PI operations in writes (generations and verifications) than reads (only verifications). However, the growing ratios in read requests (2%-14%) are larger than those in writes (2%-7%).

Bandwidth is another important metric for a storage device performance. In the next experiment, we conducted several tests to demonstrate the bandwidth variations. We used the same configurations and same I/O traces as in the latency experiment. During the testing, the bandwidths were recorded and are shown in Figure 5.6(b). After observing the results, we found that the DIX-aware MD module had a bandwidth drop compared to the original MD module. Because we had set I/O queue depth to 1, the bandwidth differences are similar to the latency differences.

Bandwidth drop ratios were also used to evaluate the bandwidth drops. As with the latency experiment, the bandwidth drops ratios are larger in sequential requests (4%-20%) than in random requests (4%-5%) for the reason that PI operations have a

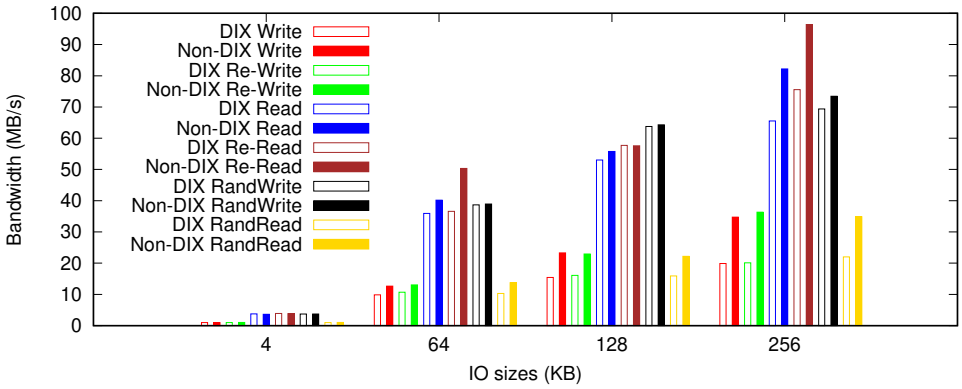


Figure 5.8: Bandwidth differences in Lustre

greater effect on the sequential request latencies. On the other hand, read requests have larger bandwidth drops ratios (3%-20%)than writes (2%-13%) because read requests have a higher latency growing ratios.

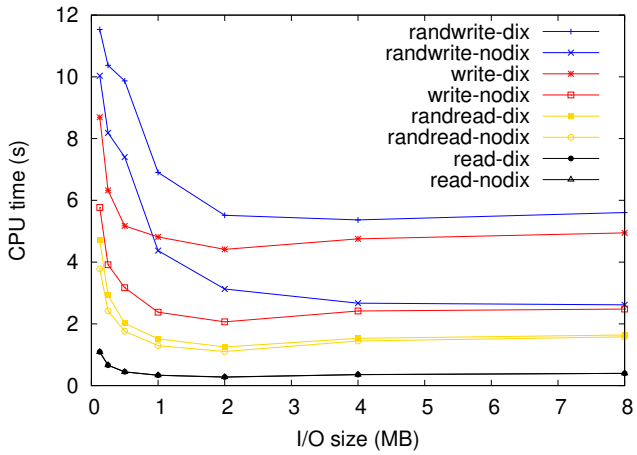


Figure 5.9: CPU time difference

In addition to greater latencies, the PI-related operations occupy extra CPU resources. In the next experiment, the CPU time consumed by the I/O requests was measured. During the execution of the same I/O traces in latency experiments, the total time consumption and the CPU utilization of the I/O requests were recorded,

which allowed us to compute the total CPU time. By comparing the CPU time between the DIX-aware MD module and the original MD module, we could estimate the CPU resources used by the PI-related operations. As shown in Figure 5.9, we found that the I/O requests in the DIX-aware MD module occupied more CPU time than in the original MD module. The increase in CPU time for read requests was insignificant because the only PI-operations were verifications. However in the write requests, where PI was both generated and verified, this led to a relatively large increase in CPU time, and the increase ratios became severe as the size of the I/O grew.

The I/O queue in the scheduler plays an important role in storage performance, especially for random requests. The I/O requests are always queued first in the I/O scheduler to optimize the performance in real scenarios. In the preceding experiments, we tested the MD module by setting the queue depth to 1, which eliminated the benefits of the I/O queue. In the following experiment, we explored how the I/O depth influenced the performance of DIX-aware MD module. The I/O depth varies from 1 ,4,16 to 32. The differences in latencies and bandwidth were calculated and are shown in Figure 5.7. By analyzing the results, we found that a higher queue depth mitigated to some degree the performance degradation, including the increases in latency and decreases in bandwidth caused by the PI operations.

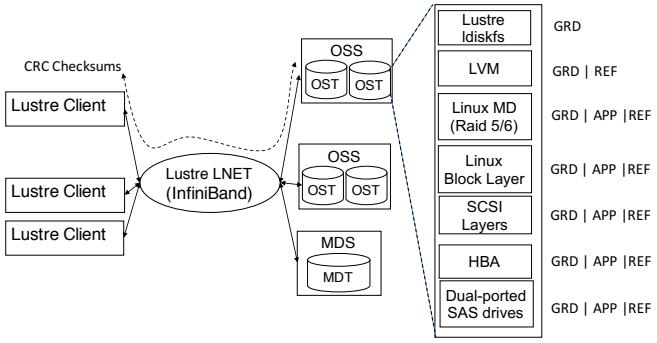


Figure 5.10: T10 DIF/DIX in Lustre

Figure 5.10 showed a design for Lustre to improve data integrity in HPC environment using T10 PI [142]. CRCs were calculated on the client, transferred over wire. On Object Storage Target (OST), the CRC checksum was attached with data sectors as GRD and finally stored on disk in writes. While in reads, the CRCs would be read

from disk, transferred over wire and checked on the client [143]. However, the current implementation for the design did not include MD RAID, which was widely deployed in Lustre for reliabilities. Therefore we installed our DIX-Aware MD module in Lustre to provide a true end-to-end data integrity in HPC and evaluated the overheads.

For the experimental environment limitations, we only deployed Lustre with one host with Metadata Server (MDS)/ Metadata Target (MDT) and Object Storage Server (OSS)/Object Storage Target (OST) and another host with a client. The Lustre version was 2.3 and Operating system and Linux kernel version were CentOS 6.3 with kernel 2.6.32. Here we used SCSI_DEBUG [144] module to simulate the PI-Capable device and IOZone benchmark at Client side with O_DIRECT IO and O_SYNC writes.

Currently, we only considered the influences from PI-Related operations on the bandwidth. We compared the bandwidth differences between the Lustre with T10 DIF/DIX support (DIX Requests) and without this support (Non-DIX Requests) in different I/O sizes (x-axis, 4KB, 64KB, 128KB and 256KB) and different I/O patterns (Write/Re-write, Read/Re-read and Random Write/Read), whose results were shown in Figure 5.8. Overall, we found that PI-related operations did bring overheads in terms of the bandwidth since the bandwidths of DIX Requests were smaller than those of Non-DIX Requests in all tests. However, in small request tests, e.g. 4KB, the bandwidth differences were not as large as those in large request tests, e.g. 128KB. The possible reason is that the small requests originally had low bandwidth, specifically low data transfer speed. Therefore, some latencies caused by PI operation did not brought significant influences on the bandwidth. This could also explain why read requests had more bandwidth decreases than writes and sequential requests had more bandwidth decreases than random requests. Based on these analysis, we believed that T10 DIF/DIX support had more impacts on bandwidth, which decreased 5% - 20% of bandwidths, in high bandwidth scenarios. While, the integrity support only brought limited negative influences on bandwidths in the low bandwidth scenarios.

5.5 Conclusion

This paper presented the design of a DIX-aware MD module design that realizes the seamless exchange of PI among end-applications running in user-mode and PI-capable

HBAs and drives. A PI operator is added to the Linux MD module to handle PI generation and verification operations. Besides, dedicated PI buffers are allocated and managed in the existed stripe structure to store and pass the PI in both write and read requests. This DIX-aware MD module improves the current DIX capabilities of the Linux I/O stack by extending it to the widely-used software RAID-5/6 personalities. Preliminary evaluations show that the DIX-aware MD module is able to detect SDCs in MD with minimal impact on performance and tolerable CPU overheads. The authors intend to contribute this implementation back to the upstream Linux kernel once it has been thoroughly tested.

Chapter 6

PMDB: A Range-based Key-Value Store on Hybrid NVM-Storage Systems

6.1 Introduction

Byte-addressable Non-Volatile Memory (NVM) provides data persistence in a memory speed [145]. Although NVM has a higher storage density than DRAM, it still may not be large enough to hold all data in this big data era [146, 147]. Therefore, this paper targets the hybrid systems, called NVM-Storage systems, including both byte-addressable NVM and block-based storage like SSD or HDD to achieve high performance and large capacity at a reasonable cost.

In the current storage infrastructure, key-value (KV) stores play an essential role in various application scenarios including data storage services [61], streaming platforms [148], machine learning pipelines [149], etc. Building KV stores on NVM-Storage systems to achieve a better performance can be very critical for upper-layer applications. Existing studies have explored and proposed several approaches [55, 53, 150, 151].

Some approaches [150, 55, 53] build KV stores on NVM-Storage systems based on Log-Structured Merge Tree (LSM-Tree) [49] to provide a high write efficiency. NVM is used as a write buffer to merge repetitive updates before writing KV pairs to storage.

However, read operations of LSM-Tree are inefficient since reading a KV pair will result in multiple storage accesses [58, 59]. Although LSM-Tree provides a higher write efficiency than other data structures, the level-based compaction of LSM-Tree introduces massive data rewrites on storage. A KV store's writing performance can be further improved by reducing data rewrites on storage [86, 60].

SLM-DB [151] builds a B+Tree index on NVM and stores KV pairs on storage in a single level. The B+Tree identifies the location of each KV pair in storage. SLM-DB achieves a good read efficiency since each read in SLM-DB only requires one storage read after searching B+Tree in NVM. However, excessive data writes on storage still occur by its selective compaction decreasing its write performance. Besides, the write performance is negatively impacted by the required large NVM space and update overhead of the B+Tree on NVM.

Although the existing approaches have explored several designs of KV stores on NVM-Storage systems, a better design is still needed to achieve good performance of both reads and writes simultaneously. To address this challenge, we propose PMDB, a range-based KV store on NVM-Storage systems that provides efficiently write and read operations.

PMDB include the following two major mechanisms: 1) a range-based data management which persists KV pairs on storage using large sequential writes with a small number of data rewrites to achieve high write efficiency; and 2) a light-weight index based on a combination of binary search tree and a novel Interval Filter Trees (IFTree) on NVM to provide a high read efficiency with limited NVM space and index update overheads. We compare PMDB with the state-of-the-art schemes including SLM-DB [151] and MatrixKV [150]. Evaluation results indicate that PMDB is the only KV store achieving both high write and read efficiencies simultaneously. In workloads with mixed reads and writes, PMDB outperforms other KV stores by 1.16× – 2.49×.

6.2 Challenges of Building Key-Value Stores on NVM-Storage Systems

Existing workload characterization [149] indicates that the performances of both read and write of KV stores are essential concerns. Therefore, we focus on building KV

stores on NVM-Storage systems to improve both read and write performances. Existing research has explored several approaches.

LSM-Tree-based Key-Value Stores LSM-Tree is a write-optimized data structure widely used in KV stores [51, 61, 152]. In LevelDB [152] implementations, LSM-Tree buffers new KV pairs in DRAM. If the buffer is full, the buffer will be flushed to Level 0 (L_0) on storage as a Sorted String Tables (SST). LevelDB stores KV pairs using multiple levels with increasing sizes based on a configurable size ratio between adjacent levels. In each level, KV pairs are stored in SSTs with disjoint key ranges. An SST consists of multiple data blocks with sorted KV pairs.

A compaction process migrates data from a smaller level to the next level if the smaller level's size reaches a threshold. For example, a compaction can include one SST from L_1 and multiple SSTs from L_2 whose key ranges overlap with the key range of the SST from L_1. The compaction is executed by reading and merging all involved SSTs in DRAM and then write them back as new disjoint SSTs in L_2. An LSM-Tree can achieve a higher write efficiency than other data structures since it only performs sequential writes. The level structure limits the total data size included in a compaction.

Several studies have built KV stores on NVM-Storage systems based on LSM-Tree [53, 55, 150]. However, they suffer from some drawbacks. Firstly, to search a key (Get), it needs to check every SST on Level 0 and an SST on each other levels until the KV pair is found or the largest level is reached. Currently, to find a key in an SST may also need multiple I/O accesses [153]. As a result, a Get operation may lead to multiple random storage I/O accesses, thus degrading the read performance significantly [58, 59]. Secondly, although the write efficiency of LSM-Tree can be higher than other data structures like B+Tree, the write efficiency can be further improved by reducing the data rewrites introduced by the leveled compaction[86, 60].

Indexing KV Pairs on NVM SLM-DB (Figure 6.1) deploys a write buffer (MemTable) in NVM, and store all SSTs on storage in a single level. Besides, SLM-DB builds a global B+Tree in NVM recording the location of every KV pair in storage. With the single-level storage, the response to a range query may be inefficient since it may access excessive overlapping SSTs. Besides, multiple versions of keys may exist in different SSTs increasing the storage space overhead. Therefore, SLM-DB deploys a selective compaction to merge overlapping SSTs in storage.

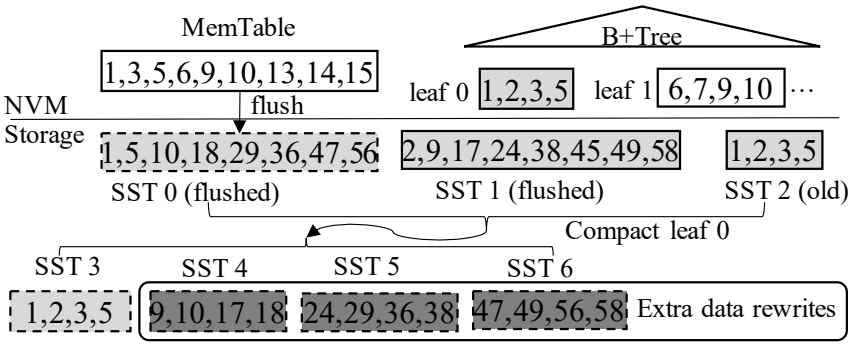


Figure 6.1: SLM-DB

SSTs will be selected as compaction candidates under two conditions: 1) If the number of distinct SSTs referenced by keys in the same leaf node of B+Tree reaches to a given maximal number, all SSTs referenced by the leaf node will be selected as compaction candidates; and 2) If the ratio of invalid keys in an SST is larger than a maximal ratio, the SST will also be selected as a compaction candidate. The selective compaction will be triggered if the number of compaction candidates reaches a threshold. SLM-DB provides a high read efficiency since a KV pair can be accessed from storage with only one storage read after searching the B+Tree to identify its location. However, the write performance of SLM-DB is impacted by the selective compaction and the global B+Tree.

First of all, the selective compaction can be triggered frequently. KV pairs in a newly-flushed SST can be distributed over a wide key range and scattered over massive leaf nodes in B+Tree. In Figure 6.1, SST 0 is a newly flushed SST from NVM and referenced by multiple leaf nodes of B+Tree, e.g., leaf 0, leaf 1, etc. One newly flushed SST can cause an increase in the SST counts of numerous leaf nodes simultaneously. As a result, a good number of SSTs will be selected as compaction candidates that triggers the selective compaction frequently.

Secondly, a selective compaction may introduce extra data rewrites by including SSTs whose key ranges are significantly different but overlapping in a small range. For instance, in Figure 6.1, SST 0 and SST 1 are newly flushed from NVM with larger sizes and wider key ranges than the existing SST 2. All these three SSTs will be selected as

compaction candidates if leaf 0 reaches its maximal count of SSTs. Then, they will be merged and rewritten during the compaction. However, the newly flushed SST 0 and SST 1 also include KV pairs with their keys out of the range of leaf 0. Moreover, those out-of-range KV pairs may be rewritten multiple times. For example, in the compaction result of SST 4, keys 17 and 18 rewritten during compacting leaf 0 will be rewritten again when compacting leaf 1.

Finally, the write performance can be further impacted by the required large NVM space and update overheads of the global B+Tree. Since the B+Tree stores every KV pair's location, its size becomes larger with an increasing number of KV pairs. With a given NVM size, the available NVM space for write buffer becomes less. A smaller write buffer may increase the total number of data writes to storage since it can absorb fewer updates before flushing out to storage. Besides, the B+Tree must be searched and updated for each KV pair when creating new SSTs during both MemTable flushes and compaction operations. The performance of searching and updating the B+Tree can be significant if we assume that NVM is multiple times slower than DRAM.

Summary LSM-Tree based KV stores suffer from inefficient read operations. The write performance can also be further improved by reducing the number of data rewrites introduced by the leveled compaction. SLM-DB achieves a good read performance by building B+Tree in NVM for every KV pair in storage. However, its write performance is impacted by the number of data rewrites on storage and large B+Tree on NVM. Therefore, our goal is to design a KV store on NVM-Storage systems which can achieve high efficiency for both write and read operations.

6.3 PMDB Overview

We learn from the existing solutions and propose PMDB. PMDB achieves high write and read efficiency based on two *Design Principles*. Firstly, PMDB will write KV pairs to storage with large sequential writes and find ways to reduce data rewrites from the required compaction. Secondly, to improve read efficiency, PMDB builds an index in NVM for data in storage. While the size of the index in NVM will be minized to avoid impacting the write performance.

6.3.1 PMDB Architecture

Figure 6.2 shows the overall architecture of PMDB. Following the *Design Principles*, PMDB includes two major mechanisms: a range-based data management to persist KV pairs with fewer data rewrites to the storage, and a light-weight NVM index with less NVM space and update overheads.

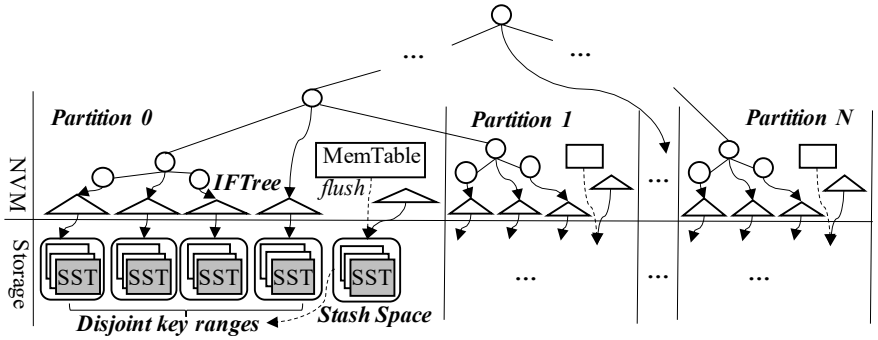


Figure 6.2: Overall architecture of PMDB

Range-Based Data Management Instead of a leveled structure, PMDB stores KV pairs on storage using small disjoint key ranges. Thus, KV pairs will not be compacted level by level and the compaction frequency can be reduced. PMDB also deploys write buffers (MemTables) on NVM. However, if we use a single large MemTable, a newly-flushed SST will overlap with a large number of key ranges on storage which may increase the data rewrites on storage in later compaction process. Besides, flushing a large MemTable may also impact the write performance [150, 154].

Therefore, as shown in Figure 6.2, PMDB partitions the disjoint key ranges and maintains a small MemTable for each partition. Each flush operation will only flush one MemTable to reduce the number of overlapped key ranges for the newly-flushed SST and avoid impacting foreground write operations significantly. In the beginning, a partition may only include one key range. MemTable needs to maintain a certain size for better I/O performance. Therefore, when KV pairs are continuously inserted, the number of disjoint key ranges will increase, but not the number of partitions. Thus, a partition may consist of several consecutive smaller key ranges.

As we discussed in Section 6.2, compacting overlapping SSTs whose key ranges are significantly wider may result in extra data rewrites. Therefore, PMDB utilizes a two-stage, partition and range, compaction to reduce the frequency of compactions. It also ensures that any compaction will only include SSTs with largely overlapped key ranges.

As shown in Figure 6.2, PMDB will flush the MemTable in a partition to a *Stash Space* as an SST. After multiple new SSTs are accumulated in the *Stash Space*, a partition compaction will be triggered to merge these overlapped SSTs. A set of new disjoint SSTs are created based on the existing disjoint key ranges in the partition as the result of compaction. Then these disjoint SSTs will be added into their corresponding key ranges. When adding a new SST to a key range, the existing SSTs in the key range will not be rewritten. Therefore, each key range may also include several overlapping SSTs. A range compaction may be triggered if a key range accumulates enough number of overlapping SSTs.

Light-Weight NVM Index PMDB uses a binary search tree to index disjoint key ranges on storage. Since both a *Stash Space* and a key range include overlapping SSTs that decrease search efficiency. To improve the search efficiency with overlapping SSTs, PMDB builds an Interval Filter Tree (IFTree) in each *Stash Space* and each disjoint key range as shown in Figure 6.2 (represented as trianglers). A node of an IFTree points to a data block in an SST. An IFTree sorts the data blocks of overlapping SSTs based on their corresponding key ranges, and stores a bloom filter for each data block.

When adding a new SST to a key range or a *Stash Space*, block information of the new SST will be added to the corresponding IFTree. When searching a key, we search a binary search tree to identify the key range, as well as the partition in storage. Then, we search the key with the order of MemTable of the partition, *Stash Space* of the partition and the key range. In the *Stash Space* or the key range, data blocks potentially including the key can be quickly identified via IFTree. Also, unnecessary storage reads can be avoided with bloom filters. An IFTree indexes the data blocks consisting of multiple KV pairs. Therefore, the total size of an IFTree is small. Besides, the false positive rate of a bloom filter is typically small. For example, in the default setting of LevelDB, the false positive rate of a bloom filter is about 0.03. The amortized number of storage reads for each search is close to one.

6.3.2 Two-Stage Compaction

The purpose of the compaction is to improve the read efficiency and delete invalid KV pairs. Therefore, PMDB sets two thresholds to trigger a compaction: the number of storage I/Os to search a key in the worst case (*max_io*) and the invalid key ratio (*invalid_ratio*). We will discuss how to set these two thresholds later. Besides, PMDB also utilizes an additional seek-based compaction to improve the performance of range queries. If range queries are executed frequently over a key range, PMDB will compact the key range's overlapping SSTs. With these compaction parameters, PMDB can be tuned to either reduce the number of rewrites or improve the read performance. The number of SSTs involved in each compaction is also reduced since a stash space or a key range will only include a limited number of overlapping SSTs.

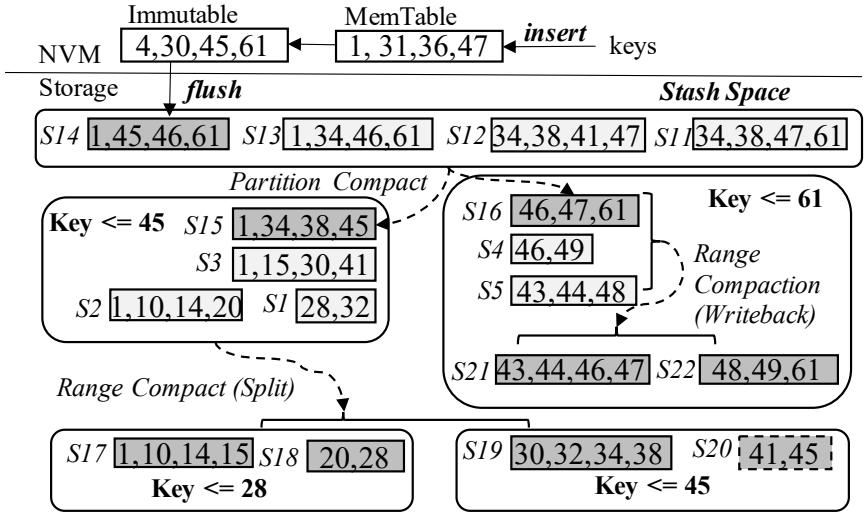


Figure 6.3: Two-stage compaction

Figure 6.3 shows the two-stage compaction process within the key range of a MemTable in a partition. We represent an SST with a format of $\{key_1, key_2, ..., key_n\}$. We also mark each SST with an SST ID, e.g. *S1*, *S2*, etc. In PMDB, an SST ID is unique and incremental. New KV pairs inserted into this partition will be first buffered in MemTable. Each MemTable is implemented as a SkipList (the same as in [53, 151]).

A MemTable will be flushed to *Stash Space* as an SST if its size reaches a threshold. Then the two-stage compaction is executed with the following process.

Partition Compaction In Figure 6.3, the *Stash Space* has four overlapping SSTs, *S14*, *S13*, *S12*, and *S11*. After a partition compaction is triggered, SSTs in the *Stash Space* will be merged. The newly generated SSTs, *S15* and *S16*, are disjoint, and created based on key range $k \leq 45$ and $45 < k \leq 61$. Finally, new SSTs are added to the corresponding key ranges without rewriting the old SSTs (i.e., data blocks in the new SSTs are updated in its corresponding IFTree).

Range Compaction A range compaction may be triggered to merge overlapping SSTs in a key range. A key range has a capacity limit (20 SSTs by default) to reduce the time and space overheads of a compaction. A range compaction has two possible operations: write back and split. The range compaction for $45 < k \leq 61$ will execute the write back operation (*Range Compaction Writeback*). New SSTs, *S21* and *S22*, are written back to the key range $45 < k \leq 61$. The compaction for the key range $k \leq 45$ performs a split (*Range Compaction Split*). Two smaller key ranges, $k \leq 28$ and $28 < k \leq 45$, are created using the median key 28 of the compaction results. The key range split does not introduce extra data rewrites since it is executed during a range compaction.

Trade-offs introduced by compaction parameters The basic principle is that the more frequent compaction is triggered, the better read and space efficiencies can be achieved, while the worse write efficiency will be. The compaction parameters, *max_io* and *invalid_ratio*, trigger compactions for different purposes. They also introduce different trade-offs among read efficiency, write efficiency, and space efficiency.

The *max_io* is set to guarantee a worst-case read performance. When *max_io is* large, a key range or a *Stash Space* may accumulate more overlapping SSTs before triggering a compaction. PMDB will have a better write efficiency since compaction is triggered less frequently. However, it decreases the read efficiency. It may also decrease space efficiency since more versions of KV pairs can exist at the same time.

The *invalid_ratio* is to improve space efficiency. If the *invalid_ratio* is small, the compaction will be triggered more frequently if there is a large number of updates. The read efficiency can also be improved since there will be fewer overlapping SSTs. However, the write efficiency will be degraded. The *invalid_ratio* may have a significant

performance influence in update-intensive workloads.

Characteristics of real-world workloads show that range queries has localities and certain key ranges may be queried more frequently [149, 155]. Therefore, PMDB also deployed a seek-based compaction. If a key range is being queried frequently, a compaction will also be triggered. The seek-based compaction improves the read, especially the range query, efficiency. Besides, the seek-based compaction limits its impact on write efficiency by only triggering compaction on frequently queried key ranges.

6.3.3 Light-Weight NVM Index with IFTrees

Building index structures for data in storage can significantly improve the read efficiency. However, the write performance can be impacted if the size of the index is too large. Therefore, building a light-weight index on NVM with small required NVM space and update overheads becomes critical. As shown in Figure 6.2, PMDB indexes key ranges using a binary search tree. A small key range and the stash space of a partition may include overlapping SSTs. IFTrees are used on NVM to index overlapping SSTs to increase the read efficiency.

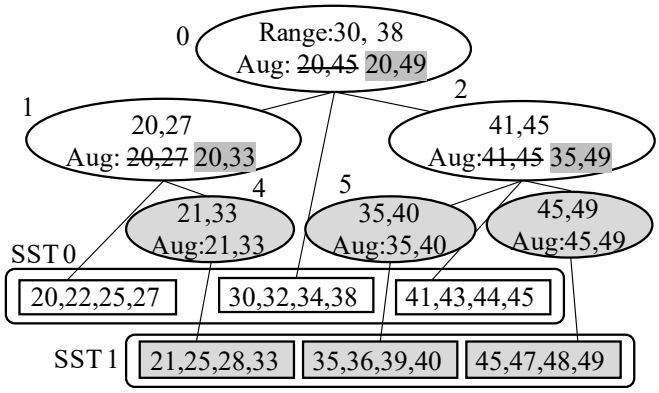


Figure 6.4: Indexing overlapping SSTs using IFTree

Figure 6.4 shows how an IFTree indexes the data blocks in overlapping SSTs. A node of IFTree includes the key range of each data block (Range) and the Augment information (Aug). Firstly, there is only *SST 0*. SST 0 has three data blocks whose

ranges are {20, 27}, {30, 38} and {41, 45}. An IFTree creates three nodes, *nodes 0, 1 and 2*, and sort them based on their start keys. The augments (Aug) of the tree nodes record the smallest and the largest keys in their subtrees. Therefore, with only SST 0, the augments of nodes 0, 1 and 2 are {20, 45}, {20, 27} and {41, 45} respectively. When *SST 1* is added including three more data blocks overlapping with those in *SST 0*. The IFTree will add three new nodes *nodes 4, 5 and 6*. The Augments of exiting nodes will be updated based on the key ranges of data blocks in SST 1.

When searching a key, subtrees of a node can be fanned out using its augment information. For example, the searching process of key 43 can start from *node 0*. The key range of *node 0* {30, 38} indicates that the indexed data block does not contain the target key 43. Before we search the children of *node 0*, we will further check its augments. The augments of *node 0* is {20, 49} which includes the target key 43. Therefore, we will further check the children, *nodes 1 and 2* of *node 0*.

However, neither the key range nor the augment of *node 1* includes the target key 43. Thus, the children of *node 1* will be removed from the follow-up searching process. Finally, we can identify that the only data block indexed by *node 2* includes the target key. Since we may search multiple overlapping data blocks for a given key, a bloom filter is also stored in the tree node to avoid unnecessary storage reads.

Since the IFTree indexes data blocks instead of each KV pair, the size of IFTree is small. PMDB can save more NVM space for write buffers. More repetitive updates can be merged in NVM further reducing the data rewrites on storage. Besides, an IFTree is also updated less frequently since it will only be updated for each data block instead of each KV pair. Therefore, the update overhead of the NVM index is smaller and can avoid impacting the write performance.

6.4 PMDB Implementation

This section discusses the implementation of PMDB including the implementation of IFTree and the range-based data management. We also introduce the interfaces of PMDB.

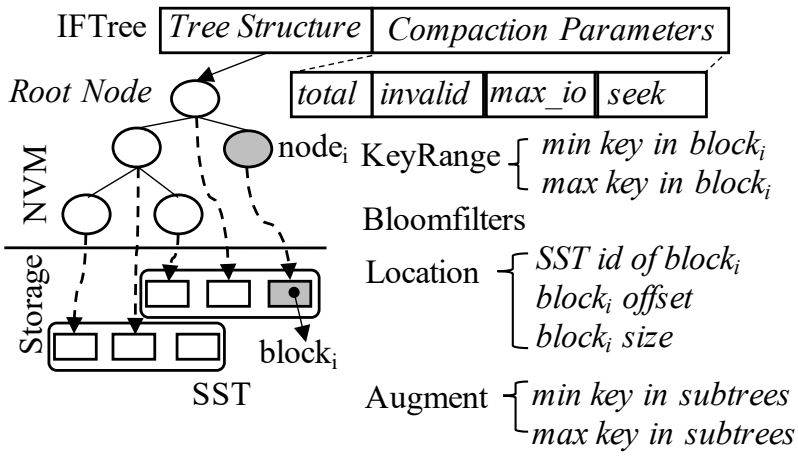


Figure 6.5: IFTree implementation

6.4.1 IFTree

IFTree is a critical data structure to index the data blocks in overlapping SSTs in a *Stash Space* and each disjoint key range. Figure 6.5 shows the implementations of IFTree. Currently, an SST includes multiple data blocks (the default block size is 4KB) [153]. Therefore, an IFTree is a tree structure with each node of the tree structure referring to a data block in an SST. Take $node_i$ (marked with dark color) as an example. To index $block_i$, $node_i$ includes four types of information, the key range of $block_i$ (KeyRange), a bloom filter of $block_i$, the location of $block_i$ and the augment.

Searching A Key The search process in IFTree starts from the root node with a Breath-First Search (BFS). When BFS arrives at $node_i$, the KeyRange and bloom filter of $node_i$ will determine if the target key exists in the data block. The node's augment including the min and max keys of the data blocks in its subtrees is used to fan out subtrees. If the target key is not in the range of augment of $node_i$, the subtrees of $node_i$ will be excluded from the follow-up BFS.

The BFS may return multiple data blocks due to the overlapping SSTs. In PMDB, SST ID is unique and incrementally increased. That means in the same *Stash Space* or a key range, the newer version key will be in the SST with a larger SST ID. Therefore,

PMDB reads the returned data blocks with a descending order of SST IDs. Same to LevelDB, PMDB can access $block_i$ from storage using the location information (SST id, offsets of $block_i$ in the SST, and the size of $block_i$).

Adding A New Data Block IFTree is a particular type of binary search tree. The nodes of IFTree are sorted based on the start keys of the indexed data blocks. When add a new $block_i$, a new $node_i$ is created. The place to add the $node_i$ can be identified by binary searching the IFTree. Since the nodes visited during this binary search are ancestors of $node_i$, the augments of visited nodes can be updated using the key range of $node_i$. A re-balance may be needed in IFTree to ensure high search efficiency. We implement IFTree by augmenting a red-block tree, a self-rebalance binary search tree [156]. Each node stores an extra bit representing color, either red or black. There are two types of re-balance operations, re-color and rotation. Node augments do not need to be modified for re-color. During a rotation, augments of the involved nodes will be updated using their new children.

Updating Compaction Parameters Both disjoint key ranges and the *Stash Space* of a partition uses an IFTree to index overlapping SSTs. Theregore, PMDB also records the parameters to trigger compactions in the IFTree. Compaction can be triggered, either in a key range or a *Stash Space* by three thresholds, *max_io*, *invalid_ratio* and *seek*. The *max_io* is the required number of storage I/Os to search a key in the worst case among the overlapping SSTs. The *invalid_ratio* is the ratio of invalid keys that can be calculated with the number of invalid keys (*invalid*) and the total number of keys (*total*). The *seek* is the number of range queries searched in the IFTree. The *max_io* and *invalid_ratio* will be updated after adding data blocks of a new SST to the IFTree.

Calculating the exact *max_io* can be difficult. A key range may include both disjoint SSTs from the last range compaction or newly added overlapping SSTs. Besides, IFTree indexes data blocks instead of every KV pair. It is almost impossible to accurately identify if a KV pair is a new insert or an update. However, PMDB does not require the exact values to trigger compactions. Therefore, we estimate *max_io* and *invalid_ratio*. In most conditions, a new SST will overlap with all existing SSTs in the key range or in a *Stash Space*. Accordingly, for each new SST, we increment the *max_io* of the IFTree. We also estimate the *invalid_ratio* using bloom filters. For each new data block, PMDB finds all of the overlapping blocks. Then PMDB checks the bloom filters of overlapping

blocks for each new key. If the bloom filter of a block indicates that the key may exist, the *invalid* is incremented. The *invalid_ratio* is calculated by *invalid* and *total*.

Space Overhead In the build-in benchmarks, dbbench [149], of LevelDB and RocksDB, the default sizes of keys, values and data blocks are 16B, 100B and 4KB respectively. A 4KB data block includes 32 KV pairs. Besides, by default, a bloom filter includes 10 bits per key. Therefore, KeyRange of a node is 32 bytes including two keys (min and max). The size of a bloom filter is 40 bytes. The location address will be 16 bytes including an 8-byte SST ID, a 4-byte block offset, and 4-byte block size. The augment of a node is also 32 bytes including two keys. Other tree structure information includes a 1-bit node color, an 8-byte left child pointer and 8-byte right child pointer. Therefore, for a 4KB data block, the size of a node of IFTree is 136 bytes, and the total size of an IFTree is about 3.32% (136B / 4KB) of the total data set. The NVM space required can be less if we use a larger block size.

6.4.2 Partitions and Key Ranges

To support Get and range queries, PMDB uses a Binary Search Tree (BST) on NVM to index disjoint key ranges in storage. Multiple key ranges are grouped by a partition and share a single MemTable.

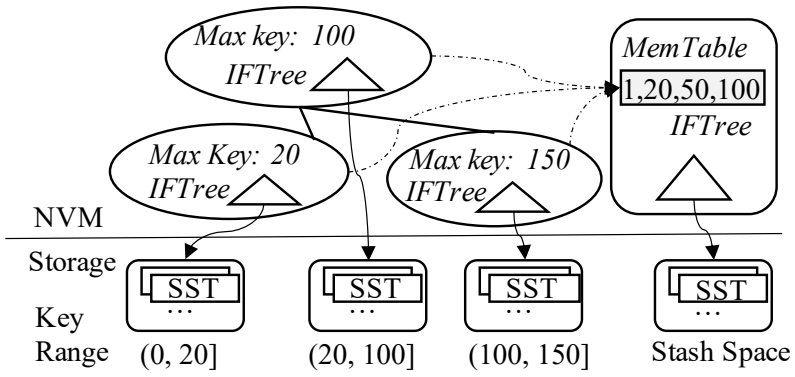


Figure 6.6: Data structure in a partition

Figure 6.6 shows the data structure in a partition. A node of a BST stores the

maximum (max) key in a key range in storage. Besides, to further improve the search efficiency on overlapping SSTs in a key range, each BST node also includes an IFTree. Key ranges in the same partition will share a MemTable. Therefore, each node of BST will include a pointer linked to the MemTable of the partition. A partition will also include an IFTree to index overlapping SSTs in *Stash Space*. When searching a key, the search process will then be executed with the order of MemTable, IFTree for the *Stash Space*, and IFTree for the key range.

PMDB creates partitions by splitting from a single partition. Thus, workloads will be balanced among different partitions if the initial portion of the workload represents the whole workload distribution. Otherwise, the partition can be initially set up based on understanding the key distribution of the workload. We will discuss the former case here.

Initially, the storage has no data and the NVM has only one Partition 0. All KV pairs will be inserted into the MemTable of Partition 0. When the MemTable of Partition 0 reaches to a maximal size, a new Partition 1 will be created using the median key in the MemTable of partition 0. Then KV pairs smaller than or equal to the median key will be copied to the MemTable of Partition 1. The partition splitting will continue until MemTables use up a predetermined NVM space. Then PMDB starts to flush MemTables to storage.

We implement BST and IFTrees using Persistent Memory Development Kits (PMDK) [157]. To ensure data consistency, *libmemobj transactions* [32] are used to modify the NVM data structure. With under-layer storage, the KV store's overall performance will be dominated by storage accesses. The overhead to ensure data consistency on NVM will not significantly impact the overall performance. We have verified that the performance overhead to ensure data consistency on NVM is not significant by comparing the overall write performance of KV store with and without transaction guarantee.

6.4.3 Key-Value Store Interfaces

As a KV store, PMDB provides following interfaces.

Put is to store a new KV pair. A new KV pair will be inserted into the corresponding MemTable in a partition. Deletions are executed using **Put** with a delete mark in the

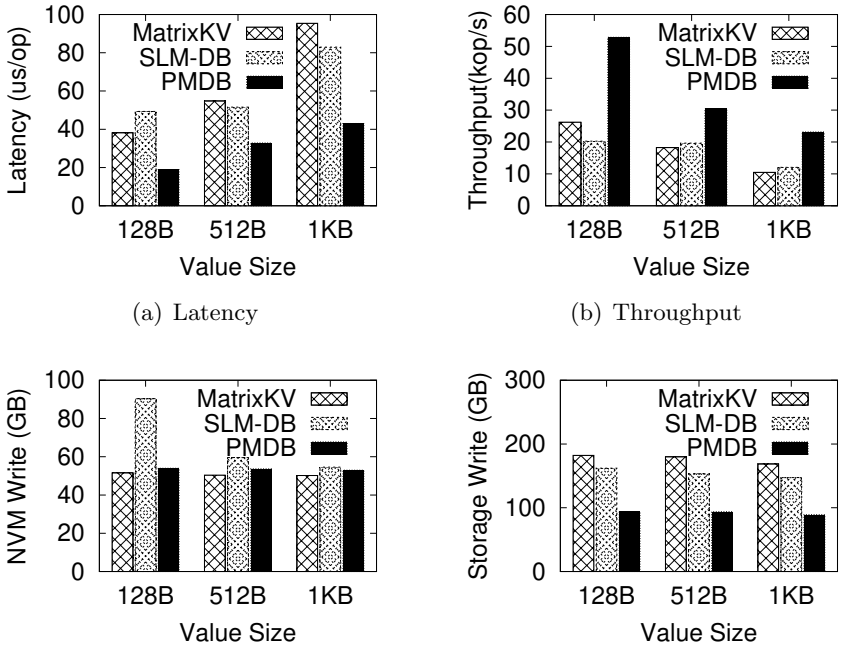
Value. Invalid KV pairs will be dropped during compaction.

Get is to search a KV pair. PMDB will search the BST to find the key range that possibly holding the KV pair. Then, it is searches the key with the order of MemTable, *Stash Space*, and the key range.

Iterator is for a range query. It consists of a *Seek* and a *Next* function. When executing a range query, PMDB performs the *Seek* using a start key and then call *Next* several times. An iterator in PMDB consists of sub-iterators of a MemTable in a partition, SSTs in *Stash Space*, SSTs in a disjoint key range. During a *Seek*, PMDB will locate the positions of the sub-iterators using the start key. For each of the *Next*, PMDB will iterate all sub-iterators simultaneously and choose the smallest key.

Recover is called after the system restarts after a failure. PMDB maintains a root object at a specific location. The root object stores the root node of BST.

6.5 Performance Evaluation



(a) Latency (b) Throughput

(c) NVM write (d) Storage write

Figure 6.7: Random write

6.5.1 Experimental Setup

Limited by the available resources, we use NVDIMM [158] and add extra delay in its write and read interfaces to emulate different NVM devices which is similar to prior works [77, 159]. We set write and read latency of the NVM to 500ns (5× DRAM) and 200ns (2× DRAM) respectively [78, 24].

We use a Serial AT Attachment (SATA) SSD to store SSTs using an ext4 file[129]. Similar with existing systems [53, 151], we implement PMDB based on LevelDB [152] with 3,329 lines of code. We compare PMDB to two state-of-the-art KV stores designed for NVM-Storage systems: MatrixKV [150] and SLM-DB [151]. We use MatrixKV to represent KV stores using LSM-Tree based structure in storage since it outperforms other designs [150, 53, 55]. SLM-DB keeps one level in storage and builds an index for each KV pair in NVM. We use two threads, one is for executing benchmarks, and another one is for background compaction. In all experiments, we disable data compression and enable bloom filters (10 bits per key). We use the default SST size (2MB), block size (4KB) and an internal block cache (8MB).

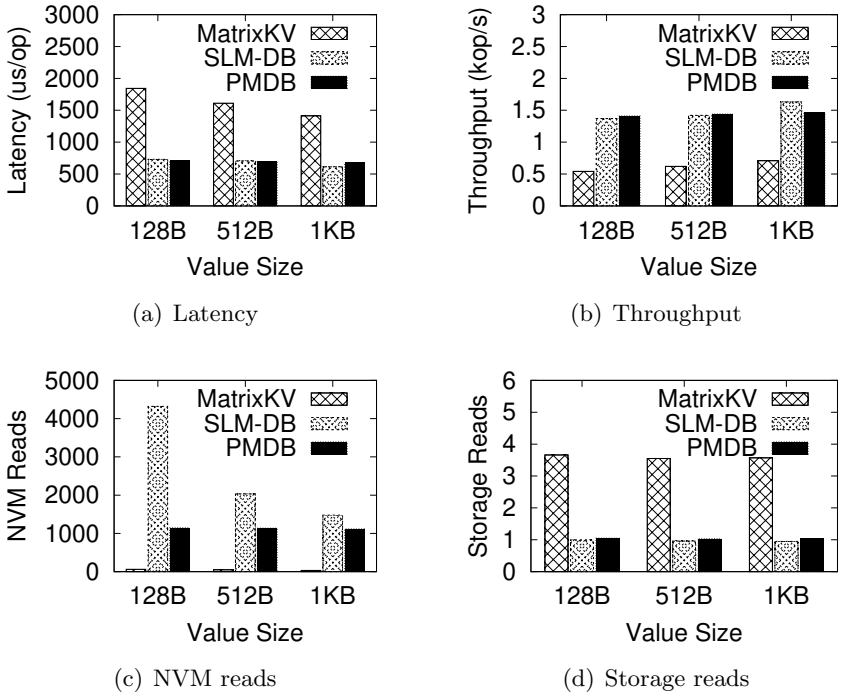
Since SLM-DB is not open-sourced, we also implement SLM-DB based on LevelDB (4,261 lines of code). In our implementation, an entry in the B+Tree is 32 bytes, including a 16-byte key and 16-byte location information (SST ID, block offset, and block size). The size of location information in SLM-DB is the same as that in PMDB. For SLM-DB, we use similar configurations in [151] including the live key ratio (0.7), the leaf node threshold (10), and the sequential degree threshold (0.8). Since SLM-DB builds an index for each KV pair, bloom filters are not needed.

In PMDB, the *max_io* and *invalid_ratio* are set to 10 and 0.3 respectively which are the same as used in SLM-DB. The seek count to trigger seek-based compaction is set to 3. The largest number of SSTs in a key range is set to 20. Besides, the default NVM partition size is the same as the maximal file size (2MB).

6.5.2 Micro-Benchmarks

We use the built-in benchmark tool, dbbench [149], to compare the performance of MatrixKV, SLM-DB, and PMDB by running different benchmarks including random

workloads and mixed skewed workloads. We also discuss the recovery and space overheads, and analyze the sensitivity of PMDB by varying the configurations.



(a) Latency (b) Throughput

(c) NVM reads (d) Storage reads

Figure 6.8: Random read

Random Workloads

Random Writes We first discuss the write performance of each design. We run random writes with 50GB total data set similar to these prior works [60, 86]. We vary the *Value* sizes among 128B (400 million KV pairs), 512B (100 million KV pairs), and 1KB (50 million KV pairs). The NVM size is 10% of the total data size close to the configurations in [53, 150].

We keep the same total NVM budget for each system. PMDB and SLM-DB will use NVM to store both index structures and MemTable. However, when the *Value* size is 128B, the size of B+Tree in SLM-DB is about 25% (32B/128B) of the total data set. The 5GB NVM is not large enough to store the B+Tree index of SLM-DB. PMDB builds an index for each data block (4KB). The index structure on NVM of PMDB is about

3% of the total data set size which can be stored in the 5GB NVM. To compare PMDB and SLM-DB with small KV pairs, in the evaluation with 128-byte *Values*, we assume the index of SLM-DB can be stored in NVM, and the write buffer size of SLM-DB is the same as that of PMDB.

Figure 6.7 shows the performance of the random writes including the latency (Figure 6.7(a)) and throughput (Figure 6.7(b)). We also measure the sizes of the total data written to NVM (Figure 6.7(c)) and storage (Figure 6.7(d)) respectively. Figure 6.7(d) indicates that PMDB achieves the smallest amount of data written to storage. Compared to MatrixKV and SLM-DB, PMDB reduces the total amount of data written to storage by 47.17% – 48.32% and 39.46% – 42.9% respectively.

Figure 6.7(c) shows that MatrixKV achieves the least amount of NVM writes since it only stores Level 0 in NVM. All compactions in MatrixKV will not generate any updates in NVM. SLM-DB and PMDB maintain index structures in NVM. The index will be updated during compactions. The total size of NVM writes in SLM-DB is significantly influenced by the number of KV pairs. With a given total data set size, a smaller *Value* means a larger number of KV pairs. Since SLM-DB stores an index entry for each KV pair, more KV pairs lead to a larger NVM writing size. PMDB only builds an index referred to each data block. Therefore, the *Value* size has less influence on its NVM writing size.

When the *Value* size is 128 bytes, the total NVM writing size, including writing to MemTable and updating the index, of PMDB is only about 4.61% higher than that of MatrixKV. Compared to SLM-DB, PMDB reduces the NVM writing size by 40.24%. The NVM writes from index updates is reduced by 90.12%. When the *Value* size is larger (512 bytes and 1KB), the NVM writing size of SLM-DB becomes smaller. PMDB still reduces the total size of NVM writes by 9.92% and 5.32% for 512-byte and 1KB *Value* sizes respectively.

However, Figure 6.7(a) and 6.7(b) show that the performance of SLM-DB is not better than that of MatrixKV with a *Value* size of 128-bytes due to the larger performance overhead for index updates. PMDB achieves the best write performance since it reduces the writing sizes to NVM and storage simultaneously. PMDB reduces the write latency with different *Value* sizes by 50.14% – 54.89% and 48.74% – 61.63% compared to those of MatrixKV and SLM-DB respectively. The throughput of PMDB is 2.02× –

2.28× and 1.92× – 2.61× higher than those of MatrixKV and SLM-DB respectively.

Random Reads We execute random read evaluations after random write experiments. Similar to [151], the number of reads covers about 20% of the total KV pairs to reduce the total execution time. We show the performance of random reads including latency (Figure 6.8(a)) and throughput (Figure 6.8(b)). Similar to [58, 59], we also measure the number of storage reads for each read operation (Figure 6.8(d)). Besides, we also measure the number of NVM reads per read to demonstrate the overhead of the index structure (Figure 6.8(c)).

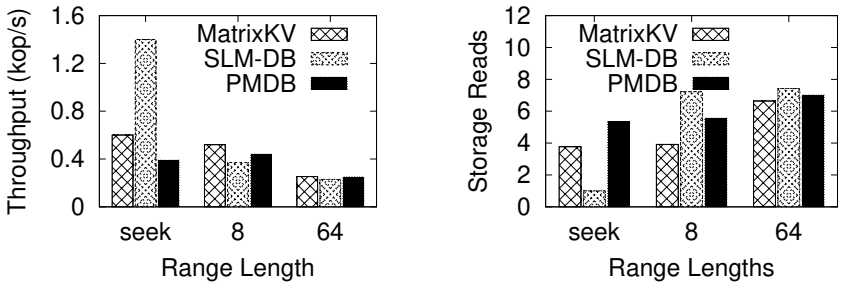
Figure 6.8(d) indicates that MatrixKV requires the most number of storage reads (3.54 – 3.66) for searching a key since a read may require to check multiple SSTs and each of SSTs needs multiple storage I/O accesses. With the global B+Tree, SLM-DB needs the least number of storage reads (0.97 – 0.99) to search a key. Although PMDB using overlapping SSTs in each key range, it can also search a key with a close efficiency to SLM-DB (1.03 – 1.05) with the help of IFTree.

Figure 6.8(c) shows that MatrixKV has the least number of NVM reads (31 – 61 per key) since it does not have an index structure in NVM. Its NVM reads are only for checking SSTs in L_0. SLM-DB has the most NVM reads since it searches the B+Tree in NVM for each key lookup. The size of B+Tree becomes larger with more KV pairs inserted. Since PMDB indexes data blocks of 4KB instead of each key, the size of IFTree is less influenced by the total number of KV pairs. Compared to SLM-DB, PMDB reduces the total number of NVM reads by 29.03% – 74.24%.

images 6.8(a) and 6.8(b) show that compared to MatrixKV, PMDB reduces the read latency by 61.54% – 51.81%. The throughput of PMDB is 2.61× – 2.06× higher than that of MatrixKV. PMDB achieves a comparable read performance to SLM-DB. When the number of KV pairs is large, e.g., with 128-byte *Value*, the read latency of PMDB can be close to that of SLM-DB since the index search in SLM-DB introduces a significant performance overhead. However, with 1KB *Value*, PMDB has a higher read latency by 11.13% than SLM-DB since the B+Tree size in SLM-DB becomes smaller compared to that of 128-byte *Value*.

Range Queries This experiment evaluates the performance of range queries. We set the *Value* to 512 bytes since the real workloads are dominated by small *Values* [149, 97]. We pre-load KV stores with 50GB data and execute three types of operations including

Seek, *Range8* and *Range64*. *Seek* means only a seek operation. *Range8* and *Range64* mean a short-range query of 8 keys and a longer range query of 64 keys respectively. Note that the start keys of range queries are randomly distributed. Therefore, seek-based compaction in PMDB will have only limited benefits.



(a) Range throughput (b) Storage reads

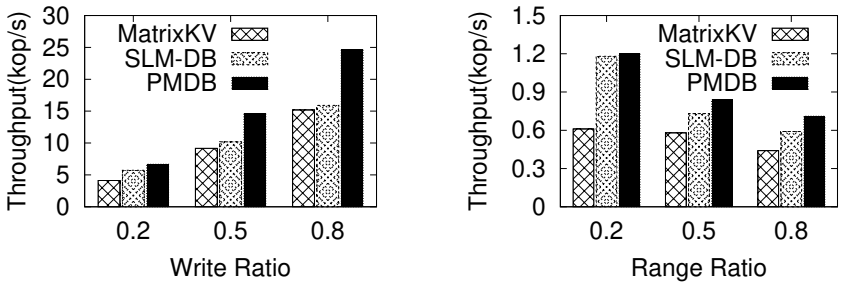
Figure 6.9: Range queries

This evaluation only shows the throughput (Figure 6.9(a)). Besides, the performance of range queries is dominated by storage I/Os since a range query requires multiple storage I/Os. Therefore, we also show the number of storage I/Os for each range query in Figure 6.9(b). Figure 6.9(b) indicates that SLM-DB achieves the highest efficiency for *Seek*. With the B+Tree, SLM-DB can execute a *Seek* with only one storage I/O. MatrixKV and PMDB need multiple storage I/Os to construct iterators with overlapping SSTs. However, a single *Seek* is less used. It is typically followed by several *Nexts*.

Range8 in Figure 6.9(b) shows the efficiency of KV stores for short-range queries. Since consecutive KV pairs may be in different SSTs, each *Next* of SLM-DB may lead to a random storage I/O. After a *Seek*, MatrixKV and PMDB have built iterators with multiple data blocks. *Next* in a short-range may not need additional storage I/Os. However, for a longer range query (*Range64*), SLM-DB can have a closer efficiency with MatrixKV and PMDB since the previously read data blocks will be cached in memory. Some *Next* can be satisfied by the cached data blocks.

Figure 6.9(a) shows that PMDB achieves a higher throughput by 16.82% for *Range8* than that of SLM-DB. For the longer range queries (*Range64*), PMDB achieves a close performance to that of MatrixKV. The throughput of *Range64* of PMDB is still 7.23%

higher than that of SLM-DB.



(a) Varying write ratios (b) Varying range ratios

Figure 6.10: Throughputs of mixed workloads

Mixed Workloads

In previous evaluations, we only discuss KV store performance in workloads with randomly generated keys (KV pairs). However, real workloads are highly skewed and include different types of operations [149, 97]. Therefore, we further discuss KV store performance in skewed workloads with mixed operations as discussed in some existing studies [149, 160].

Similar to [160], we pre-load 50GB KV pairs randomly with a *Value* size of 512 bytes. Since PMDB is less efficient for short-range queries, we especially set the length of range queries to 8. Then we run mixed workloads with different ratios of writes, reads, and short-range operations. The skewness of the workloads is defined in the same way as in [149, 160]. That is, it is the ratio of the accesses of writes, reads, and the start keys of range queries to a set of hottest keys. In our evaluation, we set the hottest key ratio and the workload skewness to 1% and 50% respectively as used in [160]. That is, 50% of total accesses are for the 1% of hottest keys.

Firstly, we vary the write ratio to represent the write-dominated scenarios (Figure 6.10(a)). We keep the same ratio of reads and range queries. For example, if the write ratio (i.e., the percentage of writes in total operations) is 0.2, both read ratio (the percentage of reads in total operations) and range ratio (the percentage of range queries in total operations) will be 0.4. Results in Figure 6.10(a) indicate that PMDB outperforms other KV stores in all workloads. The throughput of PMDB is 1.16× –

1.54× and 1.51× – 1.62× higher than those of SLM-DB and MatrixKV respectively.

Range queries influence the performance of PMDB significantly. Therefore, we further evaluate KV stores in workloads including only reads and range queries with different range ratios (Figure 6.10(b)). As shown in Figure 6.10(b), PMDB can improve the efficiency of range queries with seek-based compaction. When the range ratio is 0.2, the overall performance is dominated by reads, the throughput of PMDB is 1.96× higher than that of MatrixKV, and similar to that of SLM-DB. When the range ratio is 0.8, the throughput of PMDB is 1.21× and 1.61× of those of SLM-DB and MatrixKV respectively.

Recovery and Space Overheads

Recovery We measure the time to reopen KV stores after loading 50GB KV pairs with 512-byte *Value*. The results indicate that all KV stores can achieve fast recovery. The time required to reopen KV stores is less than one second.

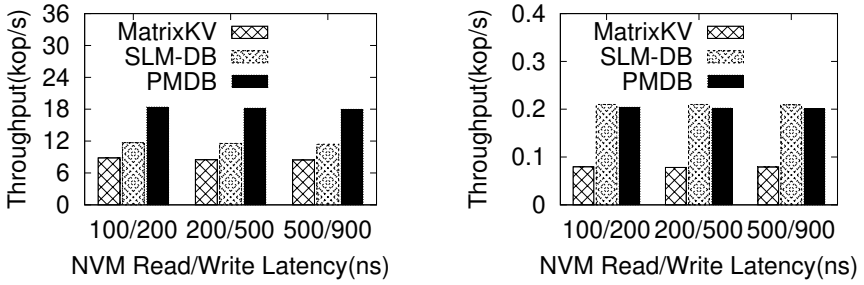
Space Overhead We measure the NVM and storage overheads after the random write workloads with 50GB KV pairs with 512-byte *Value*. The storage space for MatrixKV, SLM-DB and PMDB is 44.92GB, 45.31GB, and 45.22GB respectively. Since invalid key ratios can trigger compaction in both SLM-DB and PMDB, they do not introduce significant storage overheads than MatrixKV using leveled compaction.

The results of NVM space indicate that MatrixKV introduces the least NVM space overhead (0.08GB) since it does not build any indexes in NVM. Compared to SLM-DB (2.73GB) with indexes for every KV pair, PMDB (1.21GB) reduces the NVM space by 55.67% by constructing indexes based on data blocks. A data block can store more KV pairs if the *Value* size is small. When the *Value* size is 128 bytes, PMDB reduces the NVM space used for indexes by 88.31% compared to that of SLM-DB. The reduction of used NVM space can be more significant if we enable data compression or use larger blocks with sizes greater than 4KB.

Sensitivity Analysis

Device Performance There are multiple types of NVM with different performance [161, 151, 78]. All KV stores will execute read and write operations in NVM. The performance of NVM may impact the overall performance. Therefore, we vary NVM

latency by changing the delay time in the write and read interfaces of the reserved memory. Besides, KV stores will finally persist data in a storage device. We also run workloads in different types of storage devices besides SSDs to comprehensively study the performance influences from different storage devices.

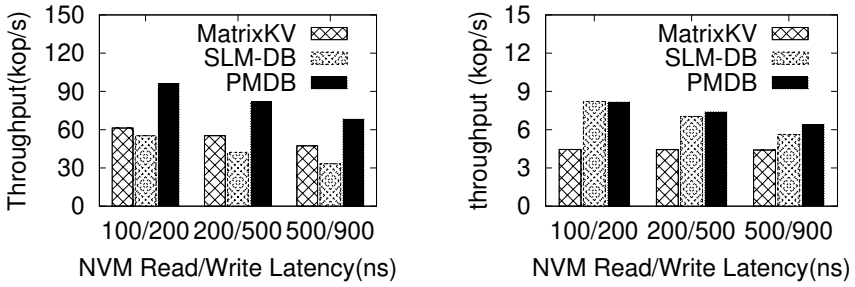


(a) Random write (b) Random read

Figure 6.11: Write/Read performance on HDD

Figure 6.11 shows the read and write throughput of three KV stores with Hard Disk Drive (HDD) as storage. Besides, we also vary the latency of NVM. However, since the latency of HDD (in milliseconds) is much larger than that of NVM (in nanoseconds) by two orders of magnitude, the overall performance will be less impacted by NVM accesses and dominated by the storage I/Os. Figure 6.11(a) shows PMDB still offers the best write performance since it reduces the number of data rewrites in storage. The throughput of PMDB is higher than those of MatrixKV and SLM-DB by 2.08× – 2.12× and 1.54× – 1.58× respectively. Figure 6.11(b) shows that PMDB still achieves a comparable read performance to that of SLM-DB. The read performance of PMDB is 2.54× – 2.56× higher than that of MatrixKV.

Figure 6.12 shows the results of varying NVM latency with a NVMe based SSD which provides a closer speed to that of NVM [162, 163]. The results indicate that the NVM latency can have more impacts on the overall performance since the performance differences between NVM and storage become smaller. The performance of MatrixKV is less impacted by NVM performance since it uses the least number of NVM writes and reads. After increasing the read/write latencies from 100/200ns to 500/900ns, the write and read performance of MatrixKV is decreased by 4.67% and 1.12% respectively. The SLM-DB performance is significantly impacted by the NVM latency since it frequently



(a) Random write (b) Random read

Figure 6.12: Write/Read performance on NVMe SSD

updates and searches the B+Tree. The write and read performance of SLM-DB are decreased by 14.52% and 19.36% respectively. The performance of PMDB is less sensitive to the changing NVM latency than that of SLM-DB since PMDB updates IFTrees less frequently. With the increases of NVM latency, the write and read performance of PMDB is decreased by 8.24% and 12.25% respectively.

NVM Space We decrease and increase the total available NVM space to 5% (5% NVM) and 20% (20% NVM) from the default 10% (10% NVM) of total size of data set respectively. Note that with 512-byte *value*, B+Tree of SLM-DB will need about 7.32% NVM to store B+Tree. Similar to the previous evaluations, we set the write buffer size of SLM-DB the same as that of PMDB and assume SLM-DB has enough NVM to store B+Tree. Then we run random writes (50GB with 512-byte *Value*) and random reads with different available NVM space. The results indicate that PMDB can still provide the best write performance with a comparable read performance to SLM-DB with different available NVM sizes. The write throughput (kop/s) of PMDB is 2.23× – 2.56× higher than those of MatrixKV and SLM-DB.

Configuration Parameters

max_io and **invalid_ratio** The max_io and invalid_ratio determine the frequency of compactions. They influence the trade-offs among write performance, read performance and storage space overhead. First of all, we keep the default invalid_ratio (0.3) and vary the max_io among 1, 10 and 20. When max_io = 1, PMDB will do read-merge-write compaction in storage. It achieves a close write and range performance to MaxtixKV and

read performance to SLM-DB respectively. When max_io = 10, the write throughput of PMDB is increased by 2.36× compared to that when max_io = 1. The random read throughput is not significantly decreased with the help of bloom filters. The performance of short-range queries is decreased by 45.26% as we discussed. When max_io = 20, the write performance of PMDB is further increased by 1.7× compared to that of max_io = 10. However, the read performance is decreased by 17.96%.

Next, we keep max_io = 10 and increases invalid_ratio from 0.3 to 0.5. The invalid_ratio has limited performance impact without enough updates. Therefore, we run random updates after loading KV pairs. After invalid_ratio is increased to 0.5 from 0.3, the write performance increases by 32.26%. However, the storage space is increased by 19.22%.

Block Size PMDB uses IFTrees referred to data blocks in a key range. We study the performance influenced by varying block sizes among 4KB, 16KB and 64KB and run random writes and reads. The results indicate that a large block size will improve the write performance. With larger blocks, the size of IFTrees becomes smaller leaving more NVM space used by write buffers. Besides, these IFTrees will be updated less frequently during a compaction. When the block sizes are 16KB and 64KB, the write throughput of PMDB is increased by 15.37% and 42.11% respectively. However, if the block size is too large, the read performance will be decreased due to the large data transfer size for one storage read. The read throughput of PMDB is decreased by 14.78% after we increase the block size from 4KB to 64KB.

Partitions We further run random writes without partitions. The whole available NVM space will be used as a single write buffer. The results indicate that the write throughput of PMDB without partitions is decreased by 21.45% compared to PMDB with partitions due to the frequently-triggered compaction.

6.5.3 Yahoo! Cloud Service Benchmark (YCSB)

Same as some of the prior work [160, 151], we further evaluate PMDB using core workloads from YCSB [99] including workloads A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads (latest values), 5% writes), E (95% ranges, 5%writes) and F (50% reads, 50% read-modify-writes). Figure 6.13 shows the throughputs of three KV stores in six core workloads. Note that the results are presented

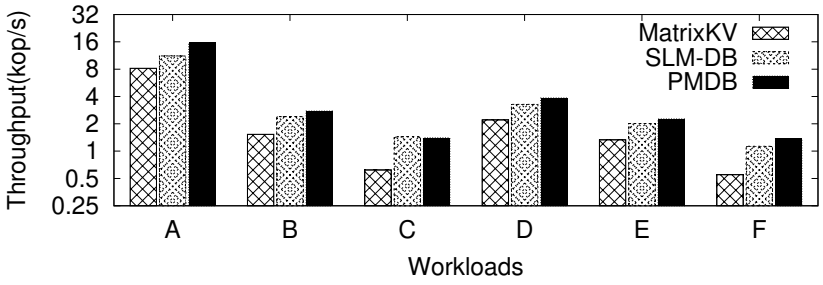


Figure 6.13: Throughput of YCSB workloads

with a log scale to make them more readable. PMDB achieves the best performance in all workloads except workload C (100% reads). In write-intensive workload A, the throughput of PMDB is 1.91× and 1.46× higher than those of MatrixKV and SLM-DB respectively. In read-intensive workloads (B, D, E, F), the throughput of PMDB is 1.27× – 2.49× and 1.15× – 1.21× higher than those of MatrixKV and SLM-DB respectively. In workload C (100% reads), the throughput of PMDB is comparable (2.95% smaller) to that of SLM-DB, and it is still 2.32× higher than that of MatrixKV.

6.6 Related Work

Write-Optimized Key-Value Stores KV stores play significant roles in big data applications [164, 84, 58]. LSM-Tree has been widely used in KV stores to serve write-intensive workloads [93, 85, 84]. Trade-offs among write, read and space amplifications in an LSM-Tree have been studied [58, 59]. Different approaches, e.g., KV separations [66, 67], new compaction methods [86, 60, 85] and etc., have been proposed.

Key-Value Stores on NVM-Storage Systems Several existing studies build KV stores for NVM-Storage hybrid systems. LSM-Tree based approaches including NoveLSM [53], NVMRocks [55], and MatrixKV [150] store a small portion of data in NVM and KV pairs in storage using a leveled structure. SLM-DB [151] stores KV pairs in storage with a single level structure and maintains a persistent B+Tree [79] index in NVM for every KV pair.

Persistent Index Structures Consistent and persistent data structures [72] including Red-Black Tree [165]. B/B+tree [80, 77, 78, 166, 167], Prefix Tree [46], Hash Table [73, 75], etc. have also been proposed for NVM.

6.7 Conclusion

In this paper, we propose PMDB, a range-based KV store for hybrid NVM-Storage systems. PMDB can provide high efficiencies for both write and read operations. We have compared PMDB to other state-of-the-art schemes designed for hybrid NVM-Storage systems. The results indicate that PMDB outperforms the other KV stores by 1.16× – 2.49×. Besides, the performance of PMDB can be tuned for either write or read-intensive workloads.

Chapter 7

Conclusion

The current storage systems are facing challenges in aspects of performance, capacity, and reliability. New storage technologies have been proposed to address the challenges above. NVM can provide data persistence in a memory speed. SMR technology increases the capacity of storage devices significantly. T10-PI is proposed to ensure end-to-end data integrity. However, the existing storage systems failed to satisfy the peculiars of the above mentioned new storage technologies. In this thesis, we target optimizing or redesigning the storage system to address their challenges by fully utilizing the latest storage technologies.

NVLSM is a key-value store using Log-Structured Merge Tree (LSM-Tree) on NVM systems. By fully leveraging the byte-addressable data accesses of NVM, NVLSM utilizes a type of accumulative compaction and a fractional cascading searching. Therefore, the write amplification can be reduced without significantly increasing the read amplification. Idler is a mechanism to control the I/O workload to minimize the tail response time of an SMR drive. Idler artificially induces idle cleaning so that the expensive blocking cleaning can be avoided. DIX-aware RAID improves the data integrity in Linux software RAID to allow PI to be verified, passed in the kernel RAID module, and stored together with data on PI-capable drives. Any data corruption, including silent data corruption, can be detected during any process of data transmission and persistence. At last, PMDB is a key-value store built on systems including both NVM and traditional storage devices to achieve performance and capacity simultaneously.

References

[1] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[2] Jiyi Wu, Lingdi Ping, Xiaoping Ge, Ya Wang, and Jianqing Fu. Cloud storage as the infrastructure of cloud computing. In *2010 International Conference on Intelligent Computing and Cognitive Informatics*, pages 380–383. IEEE, 2010.

[3] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[4] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[5] Kalev Leetaru. How do we define 'big data' and just what counts as a 'big data' analysis? `https://www.forbes.com/sites/kalevleetaru/2019/01/09/how-do-we-define-big-data-and-just-what-counts-as-a-big-data-analysis/#1d2bbae91b66`, 2019.

[6] Avantika Monnappa. How facebook is using big data - the good, the bad, and the ugly. `https://www.simplilearn.com/how-facebook-is-using-big-data-article`, 2016.

[7] Peter R Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the wafl file system. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 386–395. IEEE, 2016.

[8] Jaroslav Pokorný. Big data storage and management: Challenges and opportunities. In Jiří Hřebíček, Ralf Denzer, Gerald Schimak, and Tomáš Pitner, editors, *Environmental Software Systems. Computer Science for Environmental Protection*, pages 28–38, Cham, 2017. Springer International Publishing.

[9] Zhengyu Yang, Jiayin Wang, David Evans, and Ningfang Mi. Autoreplica: automatic data replica manager in distributed caching and data processing systems. In *2016 IEEE 35th International performance computing and communications conference (IPCCC)*, pages 1–6. IEEE, 2016.

[10] Shiuan-Tzuo Shen, Hsiao-Ying Lin, and Wen-Guey Tzeng. An effective integrity check scheme for secure erasure code-based storage systems. *IEEE Transactions on reliability*, 64(3):840–851, 2015.

[11] Moslem Didehban and Aviral Shrivastava. nZDC: A Compiler Technique for Near Zero Silent Data Corruption. In *Proceedings of the 53rd Annual Design Automation Conference*, page 48. ACM, 2016.

[12] Eduardo Berrocal, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. Lightweight Silent Data Corruption Detection Based on Runtime Data Analysis for HPC Applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 275–278. ACM, 2015.

[13] Katelin Bailey, Luis Ceze, Steven D Gribble, and Henry M Levy. Operating system implications of fast, cheap, non-volatile memory. In *HotOS*, volume 13, pages 2–2, 2011.

[14] Adam Armstrong. Methods of smr data management. http://www.storagereview.com/methods_of_smr_data_management, 2015.

[15] Technical Committee T10. Introduction to t10. `http://www.t10.org/intro.htm`, 2016.

[16] Martin K Petersen. DIF, DIX and Linux Data Integrity. *Oracle, downloaded*, page 25, 2010.

[17] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 50–61. IEEE, 2011.

[18] Simone Raoux, Feng Xiong, Matthias Wuttig, and Eric Pop. Phase change materials and phase change memory. *MRS bulletin*, 39(8):703–710, 2014.

[19] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[20] Wei Wei, Dejun Jiang, Jin Xiong, and Mingyu Chen. Hap: Hybrid-memory-aware partition in shared last-level cache. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):24, 2017.

[21] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 3–18. ACM, 2015.

[22] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on nvm. In *Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on*, pages 1–13. IEEE, 2016.

[23] Paul E McKenney. Memory ordering in modern microprocessors, part i. *Linux Journal*, 2005(136):2, 2005.

[24] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, page 1. ACM, 2013.

[25] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[26] linux. mfence. `https://www.felixcloutier.com/x86/MFENCE.html`, 2017.

[27] linux. clflush. `https://www.felixcloutier.com/x86/CLFLUSH.html`, 2017.

[28] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[29] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proceedings of the VLDB Endowment*, 10(4):337–348, 2016.

[30] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 499–512. ACM, 2017.

[31] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[32] Intel. Transactions in persistent memory development kits. `http://pmem.io/2016/05/25/cpp-07.html`, 2017.

[33] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent

memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 672–685. ACM, 2015.

[34] Alexandre P Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *Proceedings of the conference on design, automation and test in Europe*, pages 914–919. European Design and Automation Association, 2010.

[35] Jeffrey C Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm hybrid main memory. 2009.

[36] T10 Technical Committee. Information technology - zoned block commands (zbc). http://www.t10.org/ftp/zbcr01.pdf, 2014.

[37] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David HC Du. Evaluating host aware smr drives. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[38] Martin K. Petersen. I/O Controller Data Integrity Extensions. `https://oss.oracle.com/~mkp/docs/dix.pdf`.

[39] Yiming Zhang, Jinyu Zhan, Junhuan Yang, Wei Jiang, Lin Li, Li Zhu, and Xuefei Tang. Dynamic memory management for hybrid dram-nvm main memory systems. In *Embedded Software and Systems (ICESS), 2016 13th International Conference on*, pages 148–153. IEEE, 2016.

[40] Gaku Nakagawa and Shuichi Oikawa. An analysis of the relationship between a write access reduction method for nvm/dram hybrid memory with programming language runtime support and execution policies of garbage collection. In *Advanced Applied Informatics (IIAIAAI), 2014 IIAI 3rd International Conference on*, pages 597–603. IEEE, 2014.

[41] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya Dulloor. A prolegomenon on oltp database systems for non-volatile memory. *ADMS@ VLDB*, 2014.

[42] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 135–148. ACM, 2017.

[43] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Improving performance and endurance of persistent memory with loose-ordering consistency. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[44] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM SIGPLAN Notices*, volume 49, pages 433–452. ACM, 2014.

[45] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[46] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *FAST*, pages 257–270, 2017.

[47] Sanjay Ghemawat and Jeff Dean. Leveldb. `https://github.com/google/leveldb`, 2011.

[48] Facebook. Rocksdb. `URL:https://rocksdb.org/`, 2015.

[49] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[50] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: a facebook messages case study. In *FAST*, volume 14, page 12th, 2014.

[51] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[52] Jin Wang, Yong Zhang, Yang Gao, and Chunxiao Xing. Plsm: a highly efficient lsm-tree index supporting real-time big data analysis. In *2013 IEEE 37th Annual*

Computer Software and Applications Conference (COMPSAC), pages 240–245. IEEE, 2013.

[53] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 993–1005, 2018.

[54] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, 2019. USENIX Association.

[55] J Li, A Pavlo, and S Dong. Nvmrocks: Rocksdb on non-volatile memory systems, 2017.

[56] Maysam Yabandeh. Compaction in rocksdb. `https://github.com/facebook/rocksdb/wiki/Compaction`, 2018.

[57] Nadav Har'El. Scylla's compaction strategies series: Space amplification in size-tiered compaction. `https://www.scylladb.com/2018/01/17/compaction-series-space-amplification/`, 2017.

[58] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94. ACM, 2017.

[59] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520. ACM, 2018.

[60] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[61] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[62] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H.C. Du. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 129–142, Boston, MA, 2019. USENIX Association.

[63] Facebook. Rocksdb bloom filter. `http://www.lmdb.tech/bench/microbench/benchmark.html`, 2018.

[64] Intel. Persistent memory development kit. `http://pmem.io/pmdk/libpmemobj/`, 2016.

[65] Chris Mumford. Leveldb implementations. `https://github.com/google/leveldb/blob/master/doc/impl.md`.

[66] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13:5, 2017.

[67] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. Hashkv: Enabling efficient updates in {KV} storage via hashing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 1007–1019, 2018.

[68] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *ACM SIGPLAN Notices*, volume 34, pages 13–24. ACM, 1999.

[69] Songjie Niu and Shimin Chen. Optimizing cpu cache performance for pregel-like graph computation. In *Data Engineering Workshops (ICDEW), 2015 31st IEEE International Conference on*, pages 149–154. IEEE, 2015.

[70] Dean M Tullsen and Susan J Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 278–288. ACM, 1993.

[71] John Tse and Alan Jay Smith. Cpu cache prefetching: Timing evaluation of hardware implementations. *IEEE Transactions on Computers*, 47(5):509–526, 1998.

[72] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[73] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 461–476, 2018.

[74] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review*, 49(2):18–26, 2016.

[75] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 31–44, 2019.

[76] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[77] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *FAST*, volume 15, pages 167–181, 2015.

[78] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang 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*, pages 371–386. ACM, 2016.

[79] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies*, page 187, 2018.

[80] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[81] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, 2017.

[82] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[83] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–677, 1990.

[84] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: an lsm-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pages 71–82. USENIX Association, 2015.

[85] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. Slimdb: a space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.

[86] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. A light-weight compaction tree to reduce i/o amplification toward efficient key-value stores. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST'17)*, 2017.

[87] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[88] Bradley C Kuszmaul. How tokudb fractal treetm indexes work.

[89] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1303–1306. IEEE, 2009.

[90] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 217–228. ACM, 2012.

[91] Intel. pmemkv. `https://github.com/pmem/pmemkv`.

[92] Dan Williams. Persistent memory. `https://nvdimm.wiki.kernel.org/`.

[93] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. Sifrdb: A unified solution for write-optimized key-value stores in large datacenter. *ACM SoCC*, pages 477–489, 2018.

[94] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[95] HyperDex. Hyperleveldb:a fork of leveldb intended to meet the needs of hyperdex while remaining compatible with leveldb., 2016.

[96] Google. Leveldb benchmarks. `https://github.com/facebook/rocksdb/wiki/RocksDB-Bloom-Filter`, 2017.

[97] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[98] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *ACM SIGARCH Computer Architecture News*, 37(3):2–13, 2009.

[99] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[100] Intel. Core workloads in ycsb. `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`.

[101] Tim Feldman and Garth Gibson. Shingled magnetic recording areal density increase requires new data management. *USENIX; login: Magazine*, 38(3), 2013.

[102] Ming-Chang Yang, Yuan-Hao Chang, Fenggang Wu, Tei-Wei Kuo, and David HC Du. On improving the write responsiveness for host-aware smr drives. *IEEE Transactions on Computers*, 2018.

[103] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[104] Md E Haque, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, Kathryn S McKinley, et al. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM SIGPLAN Notices*, volume 50, pages 161–175. ACM, 2015.

[105] Mingzhe Hao, Gokul Soundararajan, Deepak R Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: A revelation from millions of hours of disk and ssd deployments. In *FAST*, pages 263–276, 2016.

[106] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. Pslo: enforcing the x th percentile latency and throughput slos for consolidated vm storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 28. ACM, 2016.

[107] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, pages 7–16. ACM, 2015.

[108] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[109] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight-a window on shingled disk operation. *ACM Transactions on Storage (TOS)*, 11(4):16, 2015.

[110] Linux. Kernel asynchronous i/o (aio) support for linux. http://lse.sourceforge.net/io/aio.html.

[111] Adam Manzanares Damien Le Moal. Zbc device manipulation library. https://github.com/hgst/libzbc, 2015.

[112] Chung-I Lin, Dongchul Park, Weiping He, and David HC Du. H-swd: Incorporating hot data identification into shingled write disks. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 321–330. IEEE, 2012.

[113] Chunling Wang, Dandan Wang, Yupeng Chai, and D Sun. Larger cheaper but faster: Ssd-smr hybrid storage boosted by a new smr-oriented cache framework. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST'17)*, 2017.

[114] Xuchao Xie, Liquan Xiao, Xiongzi Ge, and Qiong Li. Smrc: An endurable ssd cache for host-aware shingled magnetic recording drives. *IEEE Access*, 6:20916–20928, 2018.

[115] Microsoft Research Cambridge. Msr cambridge traces. http://iotta.snia.org/traces/388, 2008.

[116] Alireza Haghdoost, Weiping He, Jerry Fredin, and David HC Du. On the accuracy and scalability of intensive i/o workload replay. In *FAST*, pages 315–328, 2017.

[117] Adrian Palmer. Smrffs-ext4. https://github.com/Seagate/SMR_FS-EXT4, 2015.

[118] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. Hismrfs: A high performance file system for shingled storage array. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–6. IEEE, 2014.

[119] Rekha Pitchumani, James Hughes, and Ethan L Miller. Smrdb: key-value data store for shingled magnetic recording disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 18. ACM, 2015.

[120] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: a gc-free key-value store on hm-smr drives with gear compaction. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 159–171, 2019.

[121] Weiping He and David HC Du. Smart: An approach to shingled magnetic recording translation. In *FAST*, pages 121–134, 2017.

[122] Chun-Feng Wu, Ming-Chang Yang, and Yuan-Hao Chang. Improving runtime performance of deduplication system with host-managed smr storage drives. In *Proceedings of the 55th Annual Design Automation Conference*, page 57. ACM, 2018.

[123] Dongchul Park, Ziqi Fan, Young Jin Nam, and David HC Du. A lookahead read cache: improving read performance for deduplication backup storage. *Journal of Computer Science and Technology*, 32(1):26–40, 2017.

[124] Peter Macko, Xiongzi Ge, John Haskins Jr, James Kelley, David Slik, Keith A Smith, and Maxim G Smith. Smore: A cold data object store for smr drives (extended version). *arXiv preprint arXiv:1705.09701*, 2017.

[125] Adam Manzanares, Noah Watkins, Cyril Guyot, Damien Le Moal, Carlos Maltzahn, and Zvonimir Bandic. Zea, a data management approach for smr. In *HotStorage*, 2016.

[126] Swanand Mhalagi, Lide Duan, and Paul Rad. Designing and evaluating hybrid storage for high performance cloud computing. In *Systems Conference (SysCon), 2018 Annual IEEE International*, pages 1–8. IEEE, 2018.

[127] Mansour Shafaei, Mohammad Hossein Hajkazemi, Peter Desnoyers, and Abutalib Aghayev. Modeling smr drive performance. *ACM SIGMETRICS Performance Evaluation Review*, 44(1):389–390, 2016.

[128] Abutalib Aghayev, Y Theodore, Garth Gibson, and Peter Desnoyers. Evolving ext4 for shingled disks. In *FAST*, pages 105–120, 2017.

[129] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[130] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-Performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 78. IEEE Computer Society Press, 2012.

[131] Andy A Hwang, Ioan A Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *ACM SIGPLAN Notices*, volume 47, pages 111–122. ACM, 2012.

[132] What is DIF/DIX (also known as PI)? `https://access.redhat.com/solutions/41548`. Accessed 12, 2015.

[133] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[134] Jenwei Hsieh, Christopher Stanton, and Rizwan Ali. Performance evaluation of software raid vs. hardware raid for parallel virtual file system. In *Parallel and Distributed Systems, 2002. Proceedings. Ninth International Conference on*, pages 307–313. IEEE, 2002.

[135] Ziqi Fan, Fenggang Wu, Dongchul Park, Jim Diehl, Doug Voigt, and David HC Du. Hibachi: A cooperative hybrid cache with nvram and dram for storage arrays. In *Proc. of IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2017.

[136] Ziqi Fan, Alireza Haghdoost, David HC Du, and Doug Voigt. I/o-cache: A non-volatile memory based buffer cache policy to improve storage performance.

In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, pages 102–111. IEEE, 2015.

[137] Sheng Di, Eduardo Berrocal, and Franck Cappello. An Efficient Silent Data Corruption Detection Method with Error-Feedback Control and Even Sampling for HPC Applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 271–280. IEEE, 2015.

[138] Leonardo Bautista Gomez and Franck Cappello. Detecting Silent Data Corruption Through Data Dynamic Monitoring for Scientific Applications. In *ACM SIGPLAN Notices*, volume 49, pages 381–382. ACM, 2014.

[139] T10 PI in software RAID. `http://www.spinics.net/lists/raid/msg41288.html`.

[140] Enabling T10 PI in Dell PERC. `http://www.dell.com/support/manuals/us/en/19/poweredge-rc-h330/perc9ugpublication-v3/Enabling-T10-PI?guid=GUID-2CF5FB62-9984-4042-9FA1-8144761D95E0&lang=en-us`.

[141] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree Filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[142] T10-PI in Lustre. `http://www.opensfs.org/wp-content/uploads/2011/11/Improvements-in-Lustre-Data-Integrity.pdf`.

[143] Andrew Perepechko. T10-PI support implementation For Lustre. `https://review.whamcloud.com/#/c/5993/`.

[144] Douglas Gilbert. Scsi_debug adapter driver for Linux. `http://sg.danny.cz/sg/sdebug26.html`.

[145] An Chen. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25–38, 2016.

[146] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 207–219, 2019.

[147] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758, 2017.

[148] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[149] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 209–223, 2020.

[150] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.

[151] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. Slm-db: single-level key-value store with persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 191–205, 2019.

[152] Sanjay Ghemawat and Jeff Dean. LevelDB, A Fast Key-Value Storage Library. `https://github.com/google/leveldb`, 2012.

[153] Sanjay Ghemawat and Jeff Dean. Sorted tables, LevelDB Implementation. `https://github.com/google/leveldb/blob/master/doc/impl.md#sorted-tables`, 2012.

[154] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. {SILK}: Preventing latency spikes in log-structured merge key-value stores. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pages 753–766, 2019.

[155] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. Evendb: optimizing

key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[156] Sabine Hanke. The performance of concurrent red-black tree algorithms. In *International Workshop on Algorithm Engineering*, pages 286–300. Springer, 1999.

[157] Intel. Tree map in pmdk. `https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/tree_map`, 2020.

[158] Arthur Sainio. Nvdimm: changes are here so what's next. *Memory Computing Summit*, 2016.

[159] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.

[160] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 603–615. USENIX Association, July 2020.

[161] Robert Strenz. Review and outlook on embedded nvm technologies–from evolution to revolution. In *2020 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2020.

[162] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane {SSD}. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[163] Shukai Han, Dejun Jiang, and Jin Xiong. Splitkv: Splitting {IO} paths for different sized key-value items with advanced storage devices. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[164] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. {SLIK}: Scalable low-latency indexes for a key-value store.

In *2016 {USENIX} Annual Technical Conference ({USENIX} {ATC} 16)*, pages 57–70, 2016.

[165] Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. Persisting rb-tree into nvm in a consistency perspective. *ACM Transactions on Storage (TOS)*, 14(1):1–27, 2018.

[166] Dokeun Lee, Seongjin Lee, and Youjip Won. Cawbt: Nvm-based b+ tree index structure using cache line sized atomic write. *IEICE TRANSACTIONS on Information and Systems*, 102(12):2441–2450, 2019.

[167] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.