DeNova: Deduplication Extended NOVA File System

Hyungjoon Kwon[1], Yonghyeon Cho[1], Awais Khan[2], Yeohyeon Park[1], Youngjae Kim[1,†]

[1]Dept. of Computer Science and Engineering, Sogang University, Seoul, South Korea
[2]Oak Ridge National Laboratory, TN, USA
{hishine6, yongcho, yeohyeon, youkim}@sogang.ac.kr, khana@ornl.gov

***Abstract*—This paper shows mathematically and experimentally that inline deduplication is not suitable for file systems on ultra-low latency Intel Optane DC PM devices in terms of performance, and proposes DeNova, an offline deduplication specially designed for log-structured NVM file systems such as NOVA. DeNova offers high-performance and low-latency I/O processing and executes deduplication in the background without interfering with foreground I/Os. DeNova employs DRAM-free persistent deduplication metadata, favoring CPU cache line, and ensures failure consistency on any system failure. We implement DeNova in the NOVA file system. Evaluation with DeNova confirms a negligible performance drop of baseline NOVA of less than 1%, while gaining high storage space savings. Extensive experiments show DeNova is failure consistent in all failure scenario cases.**

***Index Terms*—Non-Volatile Memory, File System, Deduplication, Consistency**

I. Introduction

Recent technological advancements have enabled a generation of ultra-low latency (ULL) non-volatile memory (NVM) devices, including Intel Optane DC PM, thus blurring the performance gap between DRAM and persistent storage [1]–[4]. The Intel Optane DC PM module, in particular, can be directly connected to the memory bus alongside DRAM, exchanging data through the iMC. Furthermore, there is an XPController inside, which translates small accesses to larger accesses and thus provides comparable write latency to DRAM. For instance, the Intel Optane DC PM module has a minimum write latency of 60 ns, close to DRAM whereas its read latency is two to six times higher than DRAM as shown in Table I.

In the last several years, these advancements have triggered various innovations on the storage stack, including wide adoption of NVM file systems for new generation storage devices [4]–[9]. Several intelligent works have designed NVM file systems, such as PMFS [6], Strata [7], SplitFS [8], and NOVA [5] that guarantee high performance and low latency. These NVM file systems aim to minimize the software overhead incurred by traditional file systems by revisiting the storage stack. For instance, NOVA [5], a state-of-the-art NVM file system, adopts log-structured file system (LFS) approach to fully benefit hybrid memory systems employing DRAM and NVM devices together.

[†]Y. Kim is the corresponding author.

TABLE I
Read and write latency of memory devices.

Memory Device	Read (ns)	Write (ns)	Write Endurance
DRAM	$10 \sim 60$	$10 \sim 60$	10^{18}
PCM [10]	$50 \sim 300$	$150 \sim 1{,}000$	$10^8 \sim 10^{12}$
STT-RAM [10]	$5 \sim 30$	$10 \sim 100$	10^{15}
Optane DC PM [1]	$150 \sim 350$	$60 \sim 100$	$10^6 \sim 10^7$

However, in times of continuously growing data sizes from modern workloads, such as neural networks, databases, and graph processing, these NVM file systems not only face more pressure in terms of performance and latency but also raise serious concerns with regard to storage capacity [10]. A naive approach is to expand capacity by adding additional storage devices [11]. However, such an approach directly increases the storage, hardware management, and maintenance costs. Another alternative is to apply software-based capacity optimization techniques such as data deduplication to the NVM file systems [10], [12].

Deduplication is a specialized technique to intelligently identify and delete copies of repeated data [13]–[16]. In general, deduplication is performed either inline or offline. The former processes the deduplication before the data is stored. This directly impacts the write performance, since all processes to identify and delete duplicate data are performed in the write process. On the other hand, offline first writes all the data to the storage, which is followed by the deduplication process. This does not affect write performance, but requires temporal buffer space in the storage device. In addition, offline deduplication cannot solve the write endurance problem.

NVDedup [10] and LO-Dedup [12] are state-of-the-art NVM deduplication file systems, both designed as an inline deduplication approach for PMFS [6]. NVDedup proposes workload-adaptive fingerprinting and an NVM-favored, fine-grained metadata table. NVDedup monitors the duplicate ratio of data and selects a cost-efficient fingerprinting method accordingly. Using this method, it is claimed that whenever the duplicate ratio is high, NVDedup can outperform the baseline PMFS. LO-Dedup [12] complements a previous study on NVDedup [10], though it employs two-level fingerprints, where the second-level fingerprint is only generated when the first-level fingerprint matches. It also uses a fine-grained metadata management scheme to improve performance. What these two studies have in common is that inline deduplication is adaptively performed according to the workload or the fingerprinting algorithm to minimize performance degradation, which is the fundamental problem of inline deduplication.

However, based on our observations, we claim that state-of-the-art inline deduplication has two limitations when deployed on Intel Optane DC PM. First, it is poorly suited for Intel Optane DC PM which has a write latency comparable to DRAM. In other words, as the write latency of NVM approaches DRAM, the adaptive inline deduplication methods proposed in previous studies can no longer prevent performance degradation. The deduplication process is a compute-intensive and time-consuming operation. This overhead was tolerable with conventional storage devices with high latency. However, due to the low write latency of Intel Optane DC PM the deduplication process overhead becomes severe. This outweighs the efforts NVDedup made to optimize the inline deduplication process. Second, the DRAM-based index structure needed to look up deduplication metadata consumes a considerable amount of DRAM space.

Therefore, to address the limitations mentioned above, we propose DeNova, an offline deduplication method for NVM file systems that (i) does not degrade write performance, (ii) uses DRAM-free, self-reordering deduplication metadata, and (iii) provides strong consistency based on atomic write operation and count values, with minimum NVM access. To the best of our knowledge, DeNova is the first study on a deduplication NVM file system that does not use DRAM at all for deduplication metadata management and adopts offline deduplication, which does not cause performance degradation.

Specific contributions of the present study are as follows:

- We empirically and experimentally investigate and claim that due to its latency overhead, the performance degradation of inline deduplication is not tolerable in NVM file systems with the newly emerging ultra-low latency Intel Optane DC PM.
- By using the near-DRAM latency of NVM devices, we construct a DRAM-free deduplication metadata management table, which is a combination of a hash table and doubly linked list. Each entry, consisting of a fingerprint, reference count, block address, etc., can be loaded into a CPU cache line benefiting performance and consistency.
- We use a count-based consistency scheme for the deduplication metadata. Furthermore, by having a count value for each entry of the deduplication metadata, multiple updates can be performed concurrently.
- Minimizing the time consumed to perform deduplication leads to more available resources for other foreground processes. By employing a reordering policy in the deduplication metadata, we have minimized the number of NVM accesses during lookup, without DRAM index data structures.
- We implemented DeNova on the state-of-the-art log-structured NVM file system, NOVA and evaluated it on an Intel Optane DC PM device emulated server running Linux kernel v5.1.0. Our extensive experimental evaluation and qualitative analysis showed that (i) DeNova achieves storage efficiency with performance overhead of less than 1% compared to native NOVA and (ii) DRAM-free deduplication metadata management tables provide consistent failures in all failure cases.

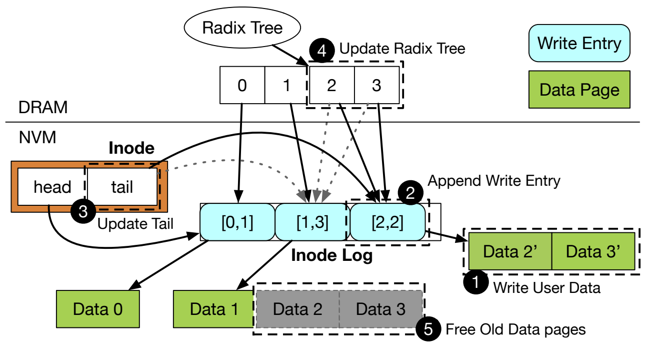


Fig. 1. Overwriting across two data pages (page 2 and 3) requires the above five steps. [filepgoff, numpages] represent the first page offset of the data pages pointed by the write entry and the number of pages written. The numbers in the radix tree represent the file page offset.

II. Background and Related Work

A. NOVA File System

The NOVA (NOn-Volatile Memory Accelerated) file system is a log-structured file system for hybrid volatile/non-volatile memory systems [5]. Although it adapts many aspects from the conventional log-structured file system, NOVA is specifically designed to exploit the fast random access of hybrid memory [17], [18]. NOVA manages a per-inode log to provide consistency. NOVA logs metadata and uses copy-on-write (CoW) on user data pages, i.e., keeping the log size small. Since multiple logs can be rapidly accessed, the per-inode log structure allows high concurrency in file accesses and recovery processes. Furthermore, NOVA keeps the per-inode log as a linked list of log pages, reducing the excessive garbage collection overhead. An invalid log page can be reclaimed without interfering with other processes. The log pages and data pages are allocated by a per-CPU memory page allocator (free list). Furthermore, NOVA uses a DRAM index data structure, radix tree, to guarantee fast access to data.

Write Flow: Figure 1 shows a write process in NOVA. ❶ NOVA first allocates a sufficient number of data pages, two data pages, to accommodate the write operation. Since NOVA uses the CoW approach, a write process always allocates new data pages. The allocated pages are filled with copied data from the user buffer and data from the previous pages that do not belong in the range of the write. ❷ A write entry is logged to the inode log, which points to the allocated pages. The write entry contains information about the first data page's file offset and the number of contiguous data pages that are allocated. Since two data pages were written starting from file page offset two, the new write entry is represented as [2,2] in Figure 1. If there is not enough space in the inode log to append a new write entry, a new log page is allocated and linked to the inode log. ❸ The inode log tail is updated to the appended write entry with an atomic 64-bit write. ❹ The index structure, radix tree, is updated to point to the new write entries accordingly and ❺ the obsolete data pages are reclaimed by the per-CPU free page list.

File System Consistency: NOVA provides consistency by using the atomic update on the inode log tail. When a system crash occurs, NOVA scans the inode log to recover the file and reconstruct the radix tree. If a system crash had occurred before appending a write entry, the allocated data pages would not be visible in the log. If a system crash had occurred after appending the write entry and before the inode tail update, the newly appended write entry would be located after the tail pointer, excluding it from the log. Therefore, in both cases the allocated data pages will not be recovered. However, if the system had crashed after the inode tail update, the write entry would be accessed and recovered. Since the write operation was either completely executed or never took place, the write process is atomic.

B. NVM File System with Deduplication

A typical deduplication process works as follows: (Step1) chunking the data, (Step2) fingerprinting the data with a hash algorithm, (Step3) looking up deduplication metadata, and (Step4) creating or updating deduplication metadata and storing unique data chunks. The deduplication process can be divided into inline and offline depending on the time of execution. The inline deduplication performs the deduplication process before data is stored to the storage [10], [12], [14], [15], [19]. Since inline deduplication is performed in the write I/O path, deduplication is inevitably accompanied by an overhead in write performance [20]. Furthermore, this overhead, mainly fingerprinting time, cannot be further reduced by software without the use of additional hardware accelerators [21]. Since deduplication is performed on DRAM before being written to NVM, it helps to improve the storage lifetime. On the other hand, the offline deduplication first writes the data and defers the deduplication process. Therefore, the offline deduplication is not accompanied by significant performance degradation. However, it requires buffer space in the storage and does not help improve write endurance.

There exist two complementary works targeting inline deduplication for file systems on persistent memory, i.e., NVDedup [10] and LO-Dedup [12]. Specifically, NVDedup [10] adopted the workload adaptive fingerprinting method to solve the problem of write performance degradation of inline-deduplication mentioned above. Through this method, NVDedup achieved minimal write performance degradation; in particular, in workloads with a high duplicate ratio, NVDedup showed even higher write performance than the original PMFS without deduplication. Similarly, LO-Dedup [12] proposed a low-overhead inline deduplication system for PMFS. Unlike NVDedup [10], LO-Dedup adopts a fast hashing scheme and sampling technique for duplication detection to minimize deduplication [12].

However, despite such performance optimization efforts as NVDedup [10] and LO-Dedup [12], the inline deduplication approach with NVM file systems cannot significantly reduce the performance overhead in real NVM devices (e.g., Intel Optane DC PM-based storage device), where its write latency is similar to DRAM write latency. Specifically, in NVDedup, the write bandwidth of NVM was assumed to be 1/8 times that of DRAM. However, the Intel Optane DC PM has a far lower write latency than that of the NVM emulation used in NVDedup. Intel Optane DC PM has a minimum write latency of 60 ns, close to DRAM, as shown in Table I. Therefore, in this paper, we argue that when targeting Intel Optane DC PM, write performance degradation due to inline deduplication in the NVM file system is very large, so we have no choice but to adopt offline deduplication rather than inline deduplication for the NVM file system.

III. Problem Definition

Traditional inline deduplication file system studies have the following two limitations.

Write Performance of Inline Deduplication with Intel Optane DC PM: To thoroughly investigate and analyze the performance of inline deduplication with NVM devices such as Intel Optane DC PM, we devised a mathematical model and validated it with experimental proofs. Table II lists all the notations used in the mathematical model.

We conducted an experiment on a 40-core machine with 64 GB of an emulated Intel Optane DC PM device. Details about the Intel Optane DC PM device emulation are shown in Section V. To this end, we used a NOVA file system with inline deduplication (DeNova-Inline). DeNova-Inline chunks the data into 4 KB, and generates a fingerprint using the SHA-1 algorithm.

We compared the time to write data (T_w) to the time taken to identify and remove duplicate data (T_f). Note that T_f consists of the data chunking time, fingerprinting time, and duplication lookup time. Each write is chunked into 4 KB, and a fingerprint is generated for each chunk. For simplification, we compare only the T_w and T_f, and exclude the additional time consumed (T_a) in the write transaction. Figure 2 shows multiple comparisons of T_w

TABLE II
Notations used in mathematical model formulation.

Notation	Description
T_w	Time consumed to write data to Intel Optane DC PM
T_f	Time required to perform chunking, fingerprinting, and duplicate lookup for strong fingerprint
T_a	Time required to complete write transaction excluding T_w and T_f
T_{fw}	Time required to perform chunking, fingerprinting, and duplicate lookup for weak fingerprint
α	Ratio of duplicates in the workload

and T_f based on the write size. With all the write sizes, the T_w never exceeds the T_f. This is due to the low write latency of NVM devices such as Intel Optane DC PM. Equation (1) models such relations between T_w and T_f.

$$T_w << T_f \tag{1}$$

The write time of a file system without inline deduplication is $T_w + T_a$, where T_a is the additional time needed to complete the write transactions. On the other hand, the write time of a NVM file system with inline deduplication is T_f + (1 - α)T_w + T_a, where α is the duplicate ratio. Every data chunk requires T_f for identifying duplicate data, and only the unique data chunks will actually be written. Equation (2) models the assumption that the performance of a file system with inline deduplication cannot exceed the performance of a file system without deduplication.

$$T_w + T_a < T_f + (1 - \alpha)T_w + T_a(0 \leq \alpha < 1) \tag{2}$$

Equation (2) is simplified to Equation (3)

$$\alpha * T_w < T_f(0 \leq \alpha < 1) \tag{3}$$

Since α is between 0 and 1, Equation (3) can be derived from Equation (1).

Furthermore, NV-Dedup [10] applies a workload adaptive fingerprinting method. It uses a weak fingerprint if the duplicate ratio is low. However, when a generated weak fingerprint matches another weak fingerprint in the deduplication metadata table, it generates a strong fingerprint to definitely identify it. Equation (4) models how the worst performance of a file system with workload adaptive inline deduplication cannot outperform the baseline file system's performance. T_{fw} indicates the time consumed to generate weak fingerprinting. The $\alpha * T_f$ indicates the worst case where all the weak fingerprints need to generate a strong fingerprint.

$$T_w + T_a < T_{fw} + \alpha * T_f + (1 - \alpha)T_w + T_a(0 \leq \alpha < 1) \tag{4}$$

Equation (4) can be simplified to Equation (5).

$$\alpha * T_w < T_{fw} + \alpha * T_f(0 \leq \alpha < 1) \tag{5}$$

Since both α and T_{fw} are bigger than 0, Equation (5) can be derived from Equation (1). Therefore, we conclude both mathematically and experimentally, that inline deduplication degrades performance in NVM file systems when deployed on Intel Optane DC PM devices.

Dedupe Metadata DRAM Space Overhead: Another limitation of existing deduplication studies lies in deduplication metadata management. The existing studies rely on DRAM either fully or partially to store the metadata [13], [22]. For instance, DmDedup [22] uses a DRAM-based CoW B-tree index that is flushed after a certain threshold. However, this would require a significant amount of DRAM space. Furthermore, in the event of a failure, recent changes cannot be reflected in the stored

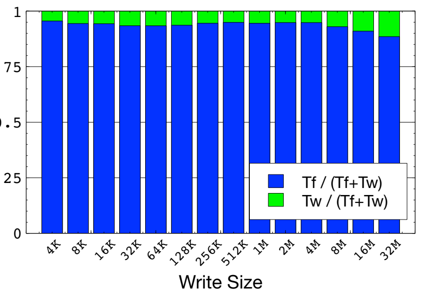


Fig. 2. Comparing the time consumed by fingerprinting (T_f) and time consumed in actual writing to the NVM device (T_w). Note that the proportion is just to compare the two times, but not the total write time. The actual write operation would also include additional operation time.

metadata unless they were flushed to the persistent storage. Another example, NVDedup [10], only uses DRAM as an indexing data structure and writes deduplication metadata directly to the NVM. This provides stronger consistency. However, the indexing data structure consumes a large amount of space in DRAM. Assuming an NVM device with N GB of storage space with 4 KB of data blocks, the indexing data structures alone can consume up to $(\frac{N\ GB}{4\ KB} \times 24\ B)/N\ GB \approx 0.6\%$ of NVM capacity in DRAM [10]. For example, a server with 1 TB-NVM and 32 GB DRAM would need 6 GB DRAM (18.75% of total DRAM capacity), just for the dedupe indexing data structures.

To solve this issue, we propose to build an offline deduplication framework for log-structured file systems such as the NOVA file system with a DRAM-free and failure-consistent PM resident deduplication metadata index data structure.

IV. DeNova File System

The motivation for DeNova is to answer a simple question: *How can we design deduplication best suitable for file systems on NVM devices with properties like an Intel Optane DC PM device?* In this section, we describe our key design principles, design, and implementation for DeNova.

A. Design Goals

- **High-performance Write I/O**: A critical constraint when designing DeNova is to maintain high I/O performance and low latency. To minimize the performance penalty for foreground write I/O, write requests for deduplication are queued and data already written is deduplicated in the background. Thus, DeNova provides an *immediate offline deduplication* mechanism that enables deduplication after data is written.
- **DRAM-Free Metadata Indexing:** Due to the explosive growth in the production of data, servers require a large amount of expensive DRAM to run their applications, which accounts for a high proportion of the server's installation and recurring costs (e.g., electricity and maintenance costs). Therefore, another important constraint is to minimize the use of DRAM required

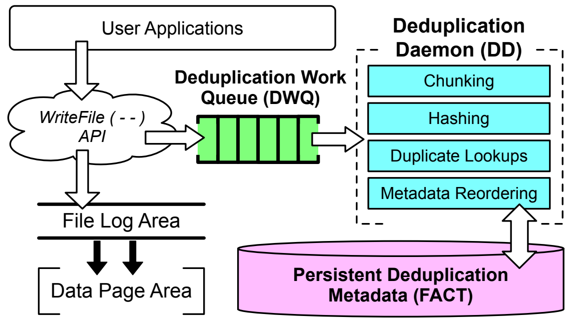


Fig. 3. An overview of DeNova.

for deduplication in NVM file systems. Accordingly, DeNova proposes a *DRAM-free deduplication metadata data structure* with high-speed access.

- **Deduplication Metadata with High Access Speed:** Since DeNova does not rely on DRAM-based indexing data structures, it needs a unique way to access a particular entry with high access speed. Furthermore, due to a higher read latency of NVM such as Intel Optane DC PM as compared to DRAM, searching algorithms on the NVM are inefficient. Therefore, DeNova is further optimized o (i) use both hash and linked list data structures with a deduplication metadata table and (ii) perform conditional reordering on the linked lists.
- **Provide Consistency with Minimum Overhead:** An unclean dismount such as a system crash can occur at any time. When a system crash occurs in the middle of a deduplication process or a reclaiming process, the reference count of a data chunk might not remain consistent, leading to loss of data and contamination of file system consistency. To prevent such cases, DeNova adapts a count-based consistency method to prevent such cases. Furthermore, every update to the deduplication metadata is followed by a flush operation to enforce ordering. This might lead to a considerable overhead. DeNova generates each metadata entry to fit into the CPU cache line to minimize the flush operations.

B. Overview

Figure 3 shows an architectural overview of DeNova. It mainly consists of several components, including a deduplication work queue (DWQ), deduplication daemon (DD), and a persistent deduplication metadata data structure called failure atomic consistent table (FACT).

1) Deduplication Work Queue: The DWQ is a dynamic first-in-first-out (FIFO) queue maintained in DRAM, where each node in the DWQ holds write request information to be deduplicated. During the write path, after appending the write entry to the file log in NOVA, the corresponding DWQ node is enqueued. Multiple writing threads can compete with each other since they share the DWQ in the write path. However, since the time spent to enqueue a node to the DWQ is extremely small as compared to the time spent accessing NVM, the bandwidth degradation is negligible. This will be discussed further in Section V. On a normal shutdown, the entries in the DWQ are saved to NVM and restored to DRAM after power on. If a system failure occurs, the DWQ is rebuilt by doing a fast scan on write entries. The dedupe-flag inside the write entries indicates the candidates for deduplication. This is further explained in Section IV-D.

2) Deduplication Daemon: The DD is a single threaded daemon service that runs in the background. Its main functions are to i) dequeue nodes from the DWQ and perform deduplication, and ii) reorder the FACT table for faster access. In the deduplication process, the DD de-queues a node from the DWQ and reads the data pages pointed by the write entry. After chunking it to 4 KB, DD generates a fingerprint using the SHA-1 hashing algorithm. Using the generated fingerprint, DD detects duplicate data chunks by looking them up in the FACT. If a duplicate data chunk is detected, deduplication is performed. Another main service of DD is the reordering of the deduplication metadata table, FACT. The reordering of FACT is described in Section IV-E. There are two variables in DD that are tunable, i.e., (n, m). Each value indicates the time interval (n msec) between each triggering point and the number of DWQ nodes (m) processed each time. When n is set to 0, the DD aggressively polls the DWQ and immediately executes the deduplication process whenever a node is enqueued in the DWQ.

3) FACT: The FACT data structure is a DRAM-free persistent deduplication metadata table containing fingerprints, data page addresses, reference count, and count-based consistency. All these important attributes ensure deduplication metadata consistency and support recovery in the event of a failure. Details about FACT are provided in Section IV-C.

C. Failure Atomic Consistent Table (FACT)

FACT is a persistent, failure-atomic, consistent, DRAM-free deduplication index metadata table. The conventional approach to improve faster lookup speed is to partially maintain a deduplication metadata table or an indexing data structure in DRAM. However, DRAM comes with limited capacity and is a costly resource. Therefore, we propose to design FACT as a static linear table to provide fast access without an additional DRAM index and store it in NVM with high storage density.

Figure 4 provides a schematic overview of FACT. Each FACT entry (64 Bytes) corresponds to a data block on NVM and consists of a number of fields or attributes. The first and second field stores the reference count (RFC) (4 B) and update count (UC) (4 B). The RFC represents the number of write entries that are pointing to the corresponding data block. Note that on mount, NOVA sets the block size to the default of 4 KB. The UC indicates the number of deduplication transactions currently targeting the corresponding data block. A modern 64-bit processor provides a 64-bit write to be atomic [6]. By using an atomic update, decreasing the UC and increasing the RFC, the deduplication process provides consistency. We discuss how FACT ensures consistency during the deduplication

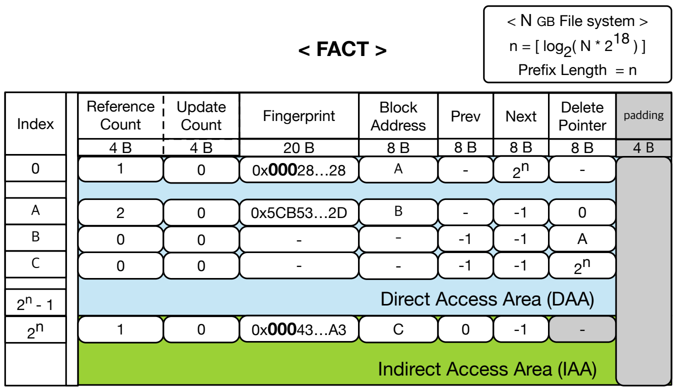


Fig. 4. A schematic representation of the FACT data structure.

process in Section IV-D. The third field stores the fingerprint (FP) (20 B). DeNova uses the SHA-1 algorithm to generate the 20 B FP. The fourth field stores the block address of the corresponding data block.

Partitioning FACT into DAA and IAA: The FP length, generated by SHA-1 in DeNova, is 160-bits. If FACT entries are indexed with the whole 160-bits, it is an ideal configuration in terms of average FACT access latency. But FACT would have 2^{160} entries, which exceeds NVM capacity. Therefore, FACT is divided into two areas: a direct access area (DAA) and indirect access area (IAA). FACT uses the prefix of FP as an index to access an entry in FACT. The prefix length (n) of FP is set to be much smaller than 160. The DAA area is accessed using the prefix of the FP, and the IAA area is accessed when the FP's prefix conflict occurs.

An entry in DAA can be accessed without additional hashing or indexing data structures, resulting in reduced average FACT access latency. However, in cases where there are two different FPs with an identical prefix, a collision might occur. This is why IAA is needed. The new entry that generated the collision is allocated in the IAA. The allocated entries with the same prefix are connected with a doubly linked list. Simply put, the DAA is a hash table and IAA is used to save buckets for hash collision, and the key is the prefix of the FP.

The fifth and sixth fields of FACT, i.e., the prev and next fields, are used to build the linked list. When a lookup is made using an FP in the deduplication process, the prefix of the FP is used as an index to read an entry from FACT. If the entry is not empty, the DD compares the FP used for lookup and the FP already saved in the FACT entry. If the values do not match, it moves on to the entry pointed by the next field. If no matches are found even after going through all the linked entries, the DD determines that the data chunk is unique and appends it to the linked list. Therefore, all the entries that are linked in the same linked list have the same FP prefix. The prev and next fields are set to -1 in initialization.

Setting the size of FACT: Since DAA needs exactly one access to NVM, while IAA needs multiple accesses to NVM, increasing the size of the DAA decreases the average FACT access latency. The size of the DAA is proportional to the length of the FP prefix. Therefore, maximizing the length of the prefix minimizes the average FACT access latency. However, as we mentioned earlier, we cannot just increase the value of n. This is because the NVM footprint of FACT increases. Therefore, configuring the prefix length (n) is crucial in the design of FACT.

DeNova sets the value of n and sets the size of the DAA and IAA areas according to the principle explained as an example below. Assuming an NVM device with N GB of storage space with 4 KB of data blocks, the minimum number of entries required in the FACT is $(\frac{N\ GB}{4\ KB} =) N{*}2^{18}$. This number is determined for the worst-case scenario where there are no duplicate data chunks. We have configured $n = \lceil \log_2(N * 2^{18}) \rceil$, to make the DAA able to comprise all $N * 2^{18}$ entries. Ideally, all the unique entries would be accessed with a single NVM access. However, if there are no duplicate data chunks and all the unique data chunks have the same prefix, all except one of the entries would be saved in the IAA. Therefore, we set the IAA size equal to the DAA, which is $N * 2^{18}$ entries.

Assuming an NVM device with N GB of storage space with 4 KB of data blocks, the proposed FACT consumes $(2 \times \frac{N\ GB}{4\ KB} \times 64\ B)/N\ GB \approx 3.2\%$ of NVM capacity. While this metadata space overhead is twice as much as that of NVDedup [10], which is 1.6%, DeNova does not require any additional DRAM index data structures. NVDedup consumes an additional 0.6% of NVM capacity in DRAM, which is explained in Section III. Since the cost of DRAM is higher than that of NVM, DeNova is more cost-effective compared to NVDedup.

The last field of FACT stores the delete pointer. This field is used in the reclaiming process of a specific data chunk. When a data page is deleted in NOVA, the free_list reclaims the data page for future use. When a data chunk is being reclaimed, it checks the reference count of the data chunk in the FACT to determine whether it can actually reclaim the data chunk. If the reference count exceeds one, we should not reclaim the data chunk. In order to lookup a data chunk in the FACT, we should use the prefix of the FP. However, we do not know the FP of the data chunk when reclaiming it. Therefore, we should first read and generate an FP of the specific data chunk. Such a process would significantly slow down the reclaiming process. In order to avoid generating an FP every time the system reclaims a data chunk, the FACT keeps a delete pointer.

The delete pointer acts as an index table that maps a block address to the corresponding FACT entry's index. A FACT entry is inserted during the deduplication process and the delete pointer is set at this point as well. Assume a FACT entry corresponding to a data block with block address B is inserted, where the FACT snapshot is shown in Figure 4. The index of the saved FACT entry is A, which is the prefix of the FP generated by the contents of the block with block address B. In order to insert a delete

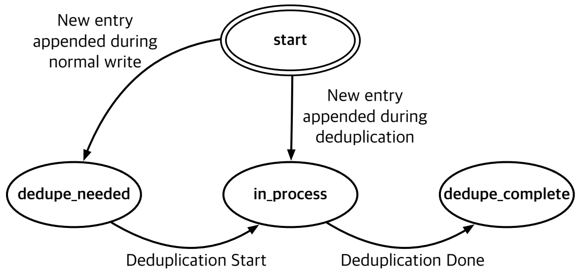


Fig. 5. State machine of the dedupe-flag, which is in the write entry. The dedupe-flag helps to provide consistency during the deduplication process.

pointer for this entry, the block address B is used as an index to set the delete pointer field. The FACT is a static linear table that is big enough to accommodate all the block indexes. Using a block address would not exceed the FACT. Therefore, A is inserted into the delete pointer field of the FACT entry, the index of which is B. Now assume that the system wants to reclaim a data block with block address B. The following three steps are required to access corresponding FACT entry of block address B. (Step 1) Use B as an index to access the FACT. In the reclaiming process, the FACT is not accessed by using the prefix of an FP as an index, but by using the block address as an index. (Step 2) Read the value saved in the delete pointer of the accessed FACT entry. In this case it would be A. (Step 3) Use A as an index to access the FACT, which is the FACT entry corresponding to block address B. After accessing the FACT entry, the reference count is decreased by one. If the reference count is 0 after the decrease, the block can be reclaimed. Since the reference count of block address B is 1 after the decrease, it is not reclaimed in this example. Simply, the delete pointer works as an indirect address. This way, the target FACT entry can be referenced from the NVM in exactly two reads from the NVM.

Lastly, the 4 B padding is added to make the FACT entry exactly fit in the CPU cache line (64 Bytes). To provide consistency between FACT entries and the actual file data, an update to an entry should be followed by a cache line flush and a memory fence instruction. Therefore, making an entry size fit into a CPU cache line reduces the maximum cache line flush and memory fence to only once, which improves performance.

D. Consistency Management

The FACT is accessed only in the following two cases: (i) to lookup FP during the deduplication process and (ii) to look up a data page during the reclaiming process. Since both cases modify the FACT and file logs, they should provide consistency in the event of system failures. In the remainder of this section, we discuss how the FACT is accessed and how DeNova provides consistency within it.

1) **Failures in Deduplication***:* The deduplication in DeNova adapts the write process of NOVA. If a duplicate data page is detected, a new write entry is appended to the log that points to the old duplicate data page. The detected duplicate data page is reclaimed afterwards. Hence, the deduplication process includes (i) updating the tail of a file log and (ii) updating the reference count of the corresponding FACT entry. To provide consistency, there should be a method to make these two updates atomic.

For instance, assume that the RFC is updated after the tail update. NOVA provides an atomic write only to update the log tail. Therefore, a system failure can occur after the tail update and before updating the RFC. Since the tail is updated, deduplication is done in terms of the file point of view. However, this transaction has not been applied to the FACT. The RFC may be smaller than the actual number of write entries pointing to the corresponding data chunk. Such a situation may lead to data loss, i.e., there is a possibility to reclaim the data page even when some file is using it. Only when the RFC is 0, the data page should be reclaimed.

2) **Deduplication Process with Consistency***:* To indicators are used by DeNova to provide consistency in deduplication.

- *update count:* As discussed earlier, this refers to the number of deduplication transactions currently in process to a specific FACT entry. At the beginning of the transaction, an atomic update increases the UC. After the transactions become persistent, an atomic update decreases the UC and increases the RFC. This prevents the inconsistency of the RFC when the transaction fails.
- *dedupe-flag:* This is an 8-bit value incorporated inside the write entry. The dedupe-flag is used not only to indicate candidates for deduplication, but also to define candidates for recovery in a system failure. The dedupe-flag has the following three states: “dedupe_needed”, “in_process”, and “dedupe_complete”. When a new write entry is appended to a normal write, its initial dedupe-flag is “dedupe_needed”. For the write entries appended during the deduplication process and currently targeted for deduplication, its dedupe-flags are set to “in_process”. When the deduplication process is finished, the dedupe-flags are set to “dedupe_complete”. The dedupe-flag is updated in place with an atomic write operation. Figure 5 shows the state transition of the dedupe-flag.

Deduplication Path: Figure 6 is a description of the deduplication process in DeNova. Most part of the deduplication process overlaps with the write process explained in Section II-A. ❶ Dequeue an entry from the DWQ. The write entry dequeued, noted as the target entry, has dedupe-flag set to “dedupe_needed.” ❷ Generate FPs for each data page and detect duplicate data chunks by checking the FACT. If a duplicate data chunk is detected, ❸ increase the UC of the corresponding FACT entry. If it is a unique data chunk, a new FACT entry is inserted with UC set to 1. In Step ❹, for each duplicate data chunk, a new write entry is appended that points to the old duplicate data page. The dedupe-flag of the appended write entry will be “in_process”. ❺ Update the log tail with an

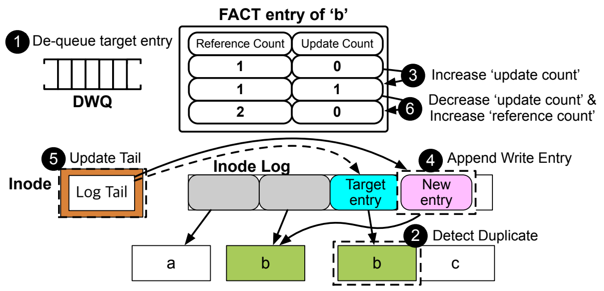


Fig. 6. An example of the deduplication process. "a","b","c" indicate the fingerprints generated by each data page. Note that only the steps that are related to FACT consistency are marked. The corresponding steps are also marked in Algorithm 1.

atomic update. After the tail is updated, the deduplication transaction is persistent to the file. Then, the dedupe-flag of the target entry is modified to "in_process". ❻ Decrease UC and increase RFC with an atomic operation. After such transactions are done, the dedupe-flag of the appended write entries, and the target entry will be set to "dedupe_complete". The obsolete duplicate data pages are reclaimed afterwards. Algorithm 1 lists the complete process.

3) **Failures in Reclaiming Process***:* Another example of inconsistency can occur in the data page reclaiming process. In NOVA, reclaiming the data page starts by adding an entry to the log and updating the log tail. This entry indicates to reclaim certain data pages. After the entry is appended, the free_list reclaims the data chunks. In DeNova an additional step to check the RFC is added in the reclaiming process. Only when the RFC is zero, its corresponding data page is reclaimed. Assume that the RFC is decreased after the tail has been updated. A failure might occur after reclaiming a data chunk but before decreasing the RFC in FACT. In such a case, the reclaimed data chunk is free to be used, but has a corresponding FACT entry filled with a wrong RFC. Such a situation results in a dangling pointer that points to the address of a reclaimed data page filled with garbage or invalid content. Therefore, for the sake of consistency, the transaction of reclaiming should take FACT into account.

E. Optimizing IAA Management

A data chunk with a high RFC is more likely to be written again. Said differently, the corresponding highly referenced FACT entry is more likely to be a target in the deduplication process. Since the IAA of FACT is managed in a doubly linked list, the access time would linearly increase with the length of the list. If a data chunk with a high reference count is located in the rear end of a linked list, this would lead to increased average NVM reads. Furthermore, the deduplication process holds an inode lock. All these would lead to a longer deduplication process time, interfering with foreground processes and wasting bandwidth of the device. The DD monitors the access time for each entry. If an entry exceeds both the predefined RFC threshold and the access time threshold,

Algorithm 1: Deduplication Algorithm

```
while $!DWQ.empty()$ do
    $target\_entry \leftarrow DWQ.pop()$; // ❶
    $inode \leftarrow get\_inode(target\_entry)$;
    $lock(inode)$;
    for $data\_page \in target\_entry$ do
        $fp \leftarrow generate\_fingerprint(data\_page)$;
        $fact\_entry \leftarrow lookup(fp)$; // ❷
        if $fact\_entry = NULL$ then
            $fact\_entry \leftarrow add\_fact\_entry(data\_page)$;
        end
        $atomic\_increase(fact\_entry.count)$; // ❸

    end
    for $data\_page \in duplicate\_data\_page$ do
        $append\_write\_entry(data\_page)$; // ❹

    end
    $inode.update\_tail()$; // ❺
    $target\_entry.dedupe\_flag \leftarrow in\_process$;
    for $write\_entry \in new\_write\_entry, target\_entry$ do
        for $data\_page \in write\_entry$ do
            $fact\_entry \leftarrow get\_fact(data\_page)$;
            $atomic\_update(fact\_entry.count)$; // ❻

        end
        $write\_entry.dedupe\_flag \leftarrow dedupe\_complete$;
    end
    $rebuild\_radix\_tree(inode)$;
    $Unlock(inode)$;
end
```

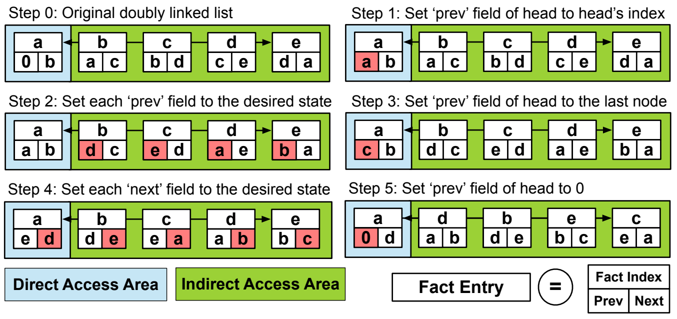
DD performs the reordering. It first holds a lock to the linked list. Then it scans the linked list and reorders them in descending order by RFC. The reordering process does not physically move the FACT entries, but only modifies the prev and next fields.

Consistency in Reordering: Reordering a doubly linked list in IAA should also consider consistency. A system crash during an in-place update could lead to a disconnected linked list. Figure 7 shows the flow of reordering. The prev field of a normal linked list head is always 0. This prev field works as a commit flag of reordering. The reordering starts by setting this prev field to the index of the head. Then it sets all the prev fields of the nodes to the desired value. After all the prev fields are updated, the commit flag is set to the last node's index. After setting the next fields of nodes to the desired value, the reordering is finished by setting the commit flag back to 0. When a system crash occurs the reordering process can be resumed or recovered by checking the commit flag. If the prev field of head is the same as its index, it uses the next field of each node to reconstruct the prev fields of each node. If the prev field is different from its index, it scans through the list using the prev fields and resumes the reordering process.

V. Evaluation

A. Evaluation Setup

We implemented DeNova in a NOVA file system and evaluated it on an NVM emulated server running Linux kernel v5.1.0. Our source code is publicly available at https://github.com/hishine6/DeNOVA. The detailed



Step 0: Original doubly linked list
Step 1: Set 'prev' field of head to head's index
Step 2: Set each 'prev' field to the desired state
Step 3: Set 'prev' field of head to the last node
Step 4: Set each 'next' field to the desired state
Step 5: Set 'prev' field of head to 0
Direct Access Area
Indirect Access Area
Fact Entry
Fact Index
Prev
Next

Fig. 7. Reordering "a-b-c-d-e" to "a-d-b-e-c" in indirect access area (IAA). Each node represents a FACT entry. In order to perform in-place updates on the NVM, there should be a method to provide consistency. By following the provided steps, the consistency of FACT can be preserved in the recovery process.

server specifications are shown in Table III. We performed experiments with two sets of synthetic workloads generated with the Fio benchmark [23], i.e., small files and large files. We also used the Fio benchmark to control the duplicate ratio in the workload.

We compare and evaluate the followings;

- **Baseline NOVA:** It denotes the NOVA file system with no deduplication.
- **DeNova-Inline:** It denotes the NOVA file system with inline deduplication. It performs all the deduplication processes (chunking, fingerprinting, duplicate lookups, dedupe metadata updates and unique data chunk storage) in the critical write I/O path.
- **DeNova-Immediate:** It denotes the NOVA file system with offline deduplication, where the DD polls DWQ aggressively and processes write entries whenever the DWQ is not empty.
- **DeNova-Delayed(n,m):** It denotes the NOVA file system with offline deduplication, where the DD is triggered every n milliseconds and consumes only m write entries from the DWQ.

Note that we designed DeNova-Inline by closely following the NVDedup [10] methodology for the NOVA file system. However, the NVDedup workload adaptive fingerprinting algorithm cannot be employed due to contradiction, as discussed in Equation (4) of Section III.

B. Performance Analysis

We first measured the write latency and deduplication latency with DeNova for small and large workload files, as shown in Table IV. In particular, we further breakdown deduplication latency into fingerprinting time and time spent for additional operations (chunking, FACT lookup, etc.). It shows that for both files, the time consumed for

TABLE III
Testbed Server Specifications.

CPU	Intel(R) Xeon(R) Gold 5218R 2.10GHz CPU Nodes (#): 2, Cores per Node (#): 40
Memory	DRAM per Node (#): 2, DDR4, 64 GB * 4 (=256 GB) (Including the PM emulated portion)
PM	Emulated on DRAM: 64 GB
OS	Linux kernel 5.1.0

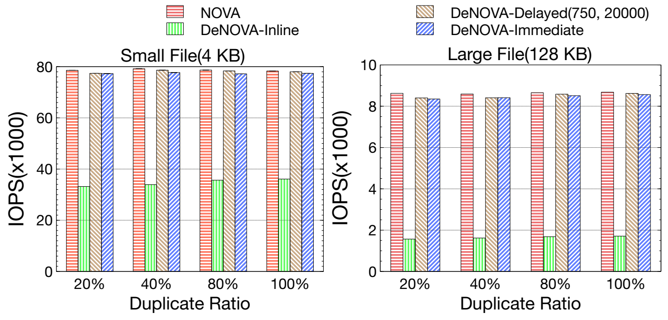


Fig. 8. Write throughput of different models, based on different duplicate ratio. DeNova-delayed(750, 20000) dequeues 20,000 write entries from the DWQ every 750 ms.

fingerprinting is five to six times longer than the write latency. With additional operations included, the total deduplication latency is six to seven times longer than the write latency.

1) Write Performance Analysis: Figure 8 shows the throughput comparison with varying duplicate ratios. For the small file workload, we used 1,000,000 files of 4 KB, and for the large file workload, we used 100,000 files of 128 KB. Also, we added 0.1 ms of think time for every 0.1 ms, which leads to a 0.2 ms cycle of think time and actual IO time. The evaluation was done with a single thread. We clearly observed a throughput drop of more than 50% for small files, and 80% for large files in DeNova-Inline compared to baseline NOVA. This is due to the large overhead of fingerprinting and the low latency of NVM devices. Although these results may not seem surprising, it proves that it completely negates the existing research results where the write throughput can be higher than the baseline NVM file system with a high duplicate ratio [10]. Note that this result also supports our claim that inline deduplication-based schemes are not the best fit for ultra-low latency NVM environments such as Intel Optane DC PM. We also observed a slight increase in performance as the duplicate ratio increased in DeNova-Inline. The actual amount written decreases as the duplicate ratio increases, leading to better performance. However, this performance increase is hard to identify in Figure 8. This is because the fingerprinting time, which does not change with different duplicate ratios, takes up most of the write transaction time. Furthermore, both DeNova-Immediate and DeNova-Delayed show a throughput drop of less than 1% compared to the baseline NOVA. This performance drop is attributed to the shared data structure, DWQ, between the write process and the deduplication process (DD).

Figure 9 shows a throughput comparison of all the variants in a multi-threaded environment. The same two

TABLE IV
File Write Latency and Deduplication Latency.

File Size		4 KB	128 KB
Write Latency (us)		2.85	39.86
Dedupe Latency (us)	Other Ops	3.66	53.57
	FP Time	11.78	215.26

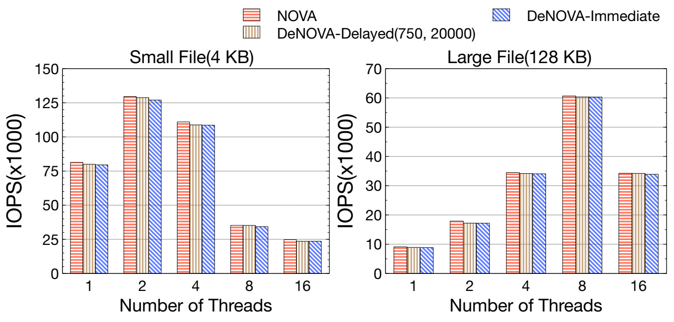


Fig. 9. Write throughput of different models, based on different numbers of threads.

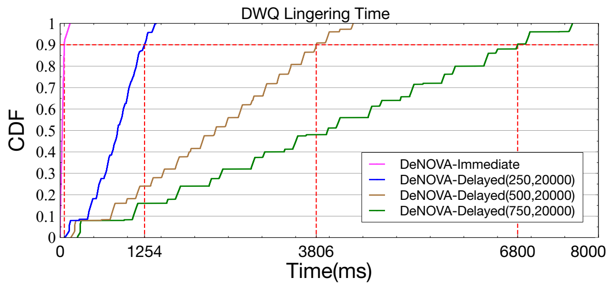


Fig. 10. Cumulative Distribution Function of the DWQ node's lingering time. For instance 90% of the DWQ nodes are processed within 1254 ms for DeNova-delayed(250,20000).

workloads from the previous experiment were used with the duplicate ratio fixed to 50%. The overall throughput increases as the number of threads increases, but after the number of threads exceeds two for small files and eights for large files, the overall throughput decreases in a parabolic pattern. We suspect that the excessive write requests of multiple threads made the file system overflow, causing this performance degradation. Overall, both DeNova-Immediate and DeNova-Delayed show the same degradation of less than 1% compared to baseline NOVA regardless of the change in the number of threads. This means that contention for the DWQ does not change significantly even when file system performance is decreased due to the increase in the number of threads.

The DeNova-Delayed(n,m) triggers the DD every n ms and processes m write entries. As n decreases it performs aggressive polling on DWQ, which is a shared data structure used by both foreground and background I/Os, possibly incurring high interference between them. Therefore, we initially expected that as n increases, the throughput would increase due to less contention on the DWQ. Simply put, we used DeNova-Delayed(n,m) to clearly demonstrate if there were any aggressive polling overhead on the DWQ. However, from previous evaluations, we proved that even when n=0 (DeNova-Immediate) the throughput degradation is small, i.e., less than 1%. The actual time accessing the DWQ takes less than 0.1% of the overall write process, and even less for the deduplication process, which takes more time compared to the write process. This leads to significantly lowering the contention possibility of DWQ. Thus, we conclude that, n does not significantly affect the throughput. On the other hand, n does affect the length of the DWQ.

2) DWQ Lingering Time and DRAM Space Overhead: To measure the average length of the DWQ, we used the average lingering time of nodes in the DWQ. The lingering time is measured by tracking the difference between each node's enqueue time and dequeue time. We used 250,000 small files (4 KB) from the previous workload. Figure 10 shows a CDF of the time of DWQ nodes lingering in the DWQ. Due to the periodic triggering of dequeue, a stair-like pattern appears in all DeNova-Delayed. As n in DeNova-Delayed(n,m) increases from 0 ms to 250 ms, the time consumed to dequeue 90% of the DWQ increases by 2,100%. The increase in the average lingering time can be interpreted as the longer average length of the DWQ. Furthermore, the DWQ is a DRAM data structure. Therefore maintaining a longer DWQ results in DRAM space overhead. Thus we conclude that, only considering the throughput and DRAM space overhead, DeNova-Immediate is the best choice.

3) Overwrite Performance Analysis: Figure 11 shows a normalized throughput comparison of write/overwrite between DeNova-Immediate and baseline NOVA. The workloads of the previous experiment were used with a single thread. In baseline NOVA, the throughput of the overwrite increases approximately 1% for large files, and 3% for small files compared to write. The write workload we used creates a new file and writes to it. This process consists of creating a new inode and allocating a new inode log page, whereas overwrite does not. Furthermore, since an inode is created per 4 KB for small files and per 128 KB for large files, the additional overhead is higher for small files. Therefore the performance increase of overwrite is slightly higher for small files. However, this phenomenon was not reproduced for DeNova-Immediate.

Since DeNova uses the CoW approach, a process of reclaiming the obsolete data page is included in the overwrite process. Furthermore, the reclaiming process in DeNova includes modifying the FACT. It should first decrease the RFC of the obsolete data page and only reclaim it when its RFC is 0. In some cases, the corresponding FACT entry could lie in the IAA. Since the IAA is maintained as a doubly linked list, when the RFC of a FACT entry in IAA is 0, the FACT entry must be removed by modifying the "prev" and "next" fields of the next and previous nodes. As mentioned in Section IV-C, a cache line flush should be followed after a FACT entry update. Therefore, a maximum of three cache line flushes ($3 * 64\ B = 192\ B$) can occur while reclaiming a single data page (4 KB). Such overhead resulted in a performance drop of approximately 5% for small files and 18% for large files in overwrites. The large file workload has a higher overhead due to higher average cache line flushes per file.

4) Evaluation of Read Performance on Duplicate File: FACT is a shared data structure. Furthermore, deduplicated data blocks are shared data pages. We performed the following experiment to evaluate whether the read thread

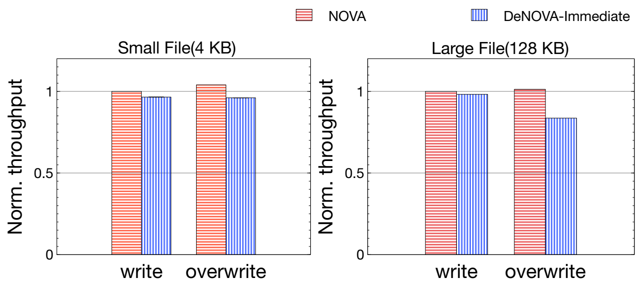


Fig. 11. Normalized throughput of write and overwrite of different models, where the write throughput of baseline NOVA is set to 1.

has performance degradation due to races on the FACT data structure or shared data page with other read or write threads.

Figure 12 shows the result of read performance in different workloads. For read-only workloads, we have made two duplicate files of a 4 GB, file A and B. We gave plenty of time in DeNova-Immediate for the DD to finish the entire deduplication process. Then we used two threads, each reading file A and B, and measured the throughput of the thread that reads file B. In NOVA, file A and B have unique data pages. On the other hand, in DeNova-Immediate, all the data pages would be shared between file A and B. Thus, the two threads would read the identical data pages. The results show no difference between NOVA and DeNova-Immediate. For the read and write mixed workloads, we started off with the read-only workloads. We used two threads, one to overwrite file A and one to read B, and measured the throughput of the thread that reads file B. There was no degradation in this case as well.

Since DeNova uses CoW, the overwritten file A would have written new data pages without interfering with the read process. Therefore, we conclude that there is no performance degradation due to shared data pages. Furthermore, since the FACT is not accessed in the read I/O path, there are no race conditions on FACT in the read process.

C. Consistency and Failure Analysis

We qualitatively analyzed the possible failures (failures during deduplication or reclaiming process) and describe the consistency handling offered by DeNova. Note that we describe the potential failures using the deduplication path discussed earlier in Figure 6 in the section IV-D.

1) Failures during Deduplication: A system crash can occur at any time during deduplication.

- **Inconsistency Handling I:** Note that, if a system failure occurs before Step 3 in Figure 6, the only thing that changes is that a write entry is dequeued from the DWQ. In system recovery, the DWQ is rebuilt by doing a fast scan on all the write entries, enqueuing all the write entries with the dedupe-flag "dedupe_needed."
- **Inconsistency Handling II:** If a system failure occurs after Step 5 and before Step 6 as shown in Figure 6, the deduplication transaction is assumed to be complete. However, the RFC needs to be updated. When scanning all the write entries at system recovery, the write entries with dedupe-flag "in_process" are detected. The detected write entries resume the deduplication process from Step 6. Even after detecting all the "in_process" write entries and finishing their deduplication process, there might be FACT entries with a UC bigger than 0. This happends when a system failure occurs after Step 3 and before Step 5. It is assumed to have failed the deduplication in such cases. Therefore, the UC is not applied to the RFC for these entries, but discarded. These UCs are set to 0 at system reboot.
- **Inconsistency Handling III:** The final case to consider is a system failure happening between Step 5 and before modifying the dedupe-flag from "dedupe_needed" to "in_process" of the target entry, which is the write entry dequeued from the DWQ. At system recovery, this write entry will be enqueued to the DWQ, even though it has been deduplicated. However, this does not cause any inconsistency issues. If Step 5 is completed, the target entry has only unique data pages and each detected duplicate data page would have already been deduplicated. For example, as can be seen in Figure 6 the target entry contains only c as a valid data page after Step 5. Since the dedupe-flag was "dedupe_needed", the UCs of these unique data pages are discarded by the recovery process. However, the UCs of the duplicate data pages, like "b", as shown in the Figure 6, would have the new write entry with dedupe-flags "in_process". This leads to a successful recovery for duplicate data pages. It shows that the deduplication process has finished for duplicate data pages but failed for unique pages. The second deduplication process done on this target entry would lead to processing deduplication only on the unique data pages.

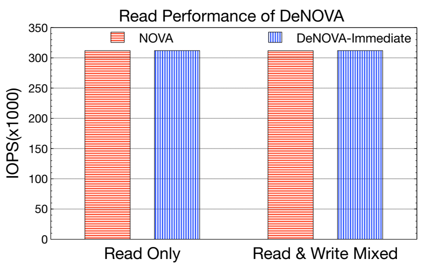


Fig. 12. Comparing the read throughput in different workloads.

2) Failures during Page Reclamation: When a system crash occurs, NOVA scans through all the write entries and generates a bitmap of occupied pages. By using this bitmap, the free_list is rebuilt, i.e., automatically finishes any reclaiming processes that were not finished. However, rebuilding the free_list does not modify the FACT. This may lead to a valid FACT entry corresponding to a free data chunk.

In order to avoid these cases, DeNova checks each FACT entry's data chunk. If the data chunk has been reclaimed by the free_list in recovery, it decreases the

RFC of the corresponding FACT entry, i.e., invalidates it. However, this recovery might lead to over increment of RFC. When a system crashes while reclaiming a data chunk with a reference count bigger than 1, rebuilding the free_list does not reclaim the data chunk. Since the data chunk is valid, the RFC does not decrease in recovery. However, the actual number of write entries pointing to the data chunk have decreased by one. While this over-increment does not affect the system consistency, it may lead to data pages that are not reclaimable. DeNova uses a background thread to monitor the use of FACT entries. It periodically scans all the files and generates a bitmap of which FACT entry is in use. If a valid FACT entry with no files using it is detected, it reclaims the corresponding data page. In this way, all the invalid data pages will eventually be reclaimed.

VI. Conclusion

This paper proposes DeNova, the most suitable deduplication framework for the latest NVM devices such as the Intel Optane DC PM module with ultra-low latency guarantees. We implemented the proposed high-performance offline deduplication framework with a DRAM-free consistent deduplication metadata index table in the NOVA file system and evaluated its effectiveness. The evaluations confirmed that DeNova shows a negligible degradation of less than 1% compared to the baseline NOVA and guarantees failure-consistency in various failure scenarios.

Acknowledgments

We thank the reviewers for their constructive comments that helped improve the paper. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. NRF-2021R1A2C2014386). This work was also supported by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

References

[1] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '20, Feb. 2020, pp. 169–182.

[2] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, April 2019, p. 427–439.

[3] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," in *ArXiv, cs.DC/1903.05714*, 2019.

[4] A. Khan, H. Sim, S. S. Vazhkudai, and Y. Kim, "MOSIQS: Persistent Memory Object Storage With Metadata Indexing and Querying for Scientific Computing," *IEEE Access*, vol. 9, pp. 85 217–85 231, 2021.

[5] J. Xu and S. Swanson, "NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '16, February 2016, p. 323–338.

[6] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System Software for Persistent Memory," in *Proceedings of the European Conference on Computer Systems*, ser. EuroSys'14, April 2014, p. 1–15.

[7] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A Cross Media File System," in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP '17, October 2017, p. 460–477.

[8] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing Software Overhead in File Systems for Persistent Memory," in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019, p. 494–508.

[9] J.-H. Kim, Y. Kim, S. Jamil, C.-G. Lee, and S. Park, "Parallelizing Shared File I/O Operations of NVM File System for Manycore Servers," *IEEE Access*, vol. 9, pp. 24 570–24 585, 2021.

[10] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue, "NV-Dedup: High-Performance Inline Deduplication for Non-Volatile Memory," *IEEE Transactions on Computers*, vol. 67, no. 5, pp. 658–671, 2018.

[11] M. Ajdari, P. Park, D. Kwon, J. Kim, and J. Kim, "A Scalable HW-Based Inline Deduplication for SSD Arrays," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 47–50, Jan 2018.

[12] W. Chen, Z. Chen, D. Li, H. Liu, and Y. Tang, "Low-overhead Inline Deduplication for Persistent Memory," *Transactions on Emerging Telecommunications Technologies*, vol. 32, no. 8, 2021.

[13] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, Inline Data Deduplication for Primary Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '12, February 2012, pp. 1–14.

[14] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: A Scalable Secondary Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '09, February 2009, p. 197–210.

[15] A. Khan, P. Hamandawana, and Y. Kim, "A Content Fingerprint-Based Cluster-Wide Inline Deduplication for Shared-Nothing Storage Systems," *IEEE Access*, vol. 8, pp. 209 163–209 180, 2020.

[16] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A Study on Data Deduplication in HPC Storage Systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 7:1–7:11.

[17] J.-H. Kim, Y. Kim, S. Jamil, and S. Park, "A NUMA-aware NVM File System Design for Manycore Server Applications," in *Proceedings of the 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '20, 2020, pp. 1–5.

[18] J.-H. Kim, J. Kim, H. Kang, C.-G. Lee, S. Park, and Y. Kim, "PNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '19, New York, NY, USA, 2019, p. 1–7.

[19] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li, "Decentralized Deduplication in SAN Cluster File Systems," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC '09, June 2009.

[20] A. Khan, C.-G. Lee, P. Hamandawana, S. Park, and Y. Kim, "A Robust Fault-Tolerant and Scalable Cluster-Wide Deduplication for Shared-Nothing Storage Systems," *Proceedings of the 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 87–93, 2018.

[21] P. Hamandawana, A. Khan, C.-G. Lee, S. Park, and Y. Kim, "Crocus: Enabling Computing Resource Orchestration for Inline Cluster-Wide Deduplication on Scalable Storage Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1740–1753, 2020.

[22] D. Jain, G. Kuenning, H. Mudd, S. Mandal, K. Palanisami, S. Trehan, and E. Zadok, "Dmdedup : Device Mapper Target for Data Deduplication," in *Proceedings of the 2014 Ottawa Linux Symposium)*, ser. OLS '14, 2014.

[23] J. Axboe, *Flexible I/O Tester*, https://github.com/axboe/fio.