NV-Dedup: High-Performance Inline Deduplication for Non-Volatile Memory

Chundong Wang, Qingsong Wei, *Senior Member, IEEE*, Jun Yang, Cheng Chen, Yechao Yang, and Mingdi Xue

Abstract—The byte-addressable non-volatile memory (NVM) is a promising medium for data storage. NVM-oriented file systems have been designed to explore NVM's performance potential. Meanwhile, applications may write considerable duplicate data. For NVM, a removal of duplicate data can promote space efficiency, improve write endurance, and potentially improve the performance by avoidance of repeatedly writing the same data. However, we have observed severe performance degradations when implementing a state-of-the-art inline deduplication algorithm in an NVM-oriented file system. A quantitative analysis reveals that, with NVM, 1) the conventional way to manage deduplication metadata for block devices, particularly in light of consistency, is inefficient, and, 2) the performance with deduplication becomes more subject to fingerprint calculations. We hence propose a deduplication algorithm called *NV-Dedup*. NV-Dedup manages deduplication metadata in a fine-grained, CPU and NVM-favored way, and preserves the metadata consistency with a lightweight transactional scheme. It also does workload-adaptive fingerprinting based on an analytical model and a transition scheme among fingerprinting methods to reduce calculation penalties. We have built a prototype of NV-Dedup in the Persistent Memory File System (PMFS). Experiments show that, NV-Dedup not only substantially saves NVM space, but also boosts the performance of PMFS by up to 2.1×.

Index Terms—Non-volatile memory, inline deduplication, NVM-oriented file system, metadata consistency, workload adaptation

1 Introduction

The development of phase change memory (PCM) [1], Resistive RAM (ReRAM) [2], spin-transfer torque RAM (STT-RAM) [3], and 3D XPoint [4], is encouraging. Unlike flash memory, the next-generation non-volatile memory (NVM) is byte-addressable, and has comparable read latency and longer write latency than DRAM. Thus it can be put onto memory bus, sitting alongside DRAM for CPU's direct load and store [5], [6], [7], [8].

NVM attracts much attentions to explore its performance potentials. NVM-oriented file systems have been developed for applications to efficiently store and access data with NVM [5], [8], [9], [10], [11]. However, data written by applications, also known as the *primary data*, may have a considerable duplication ratio (the percentage of duplicate data in all data) [12], [13], [14]. A removal of duplicate data shall significantly promote space efficiency of NVM. Fewer write operations also benefit the lifetime of NVM considering some NVM's write endurance issues [1], [15]. More important, if the time saved by an avoidance of storing duplicate data is much more than the execution time for identifying and managing duplicate data, an NVM-oriented file system should yield higher performance.

Inline deduplication for primary data has been done on legacy block-based storage devices [16], [17], [18], [19], [20]. In brief, deduplication calculates a *fingerprint* for a data chunk using a cryptographic hash function, like MD5 [21], SHA-1 or SHA-256 [22], and searches it in existing fingerprints to determine whether the chunk is duplicate or not. Duplicate data need not be stored.

Stat-of-the-art inline deduplication algorithms organize deduplication metadata, like fingerprints, in the block format. They are loaded into DRAM for use and periodically synchronized to storage device for consistency. Applying such a deduplication algorithm for NVM is unsatisfactory for two reasons. First, since NVM is byte-addressable, the block-based organization is not rational. Second, the synchronization between DRAM and NVM incurs significant performance overheads. As NVM has comparable access latency to DRAM, it is not necessary to keep dual copies of metadata in DRAM and NVM.

Although NVM supports in-place metadata management for deduplication and allows direct updating metadata without synchronization, the preservation of metadata consistency still impairs performance. In order to keep deduplication metadata consistent, logging or copy-on-write (COW) with ordered write is required for in-NVM structures. However, such consistency mechanisms may cause substantial performance penalties. In the meantime, ordinary structures, like B-tree, do not take into account the usage characteristics of deduplication metadata. As a result, fine-grained metadata management with lightweight consistency and specific

- *C. Wang is with the Singapore University of Technology and Design, Singapore 487372. E-mail: wangc.nus@gmail.com.*
- *Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue are with the Data Stoarge Institute, A*STAR, Singapore 138632. E-mail: {WEI_qingsong, yangju, Chen_Cheng, yangyc, XUE_Mingdi}@dsi.a-star.edu.sg.*

Manuscript received 13 Feb. 2017; revised 6 Nov. 2017; accepted 8 Nov. 2017. Date of publication 19 Nov. 2017; date of current version 13 Apr. 2018. (Corresponding author: QingsongWei.)
Recommended for acceptance by F. Douglis.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2774270

optimization for deduplication metadata is highly demanded for inline deduplication on NVM.

On the other hand, fingerprinting significantly impacts the performance when deduplicating data for NVM. Using a cryptographic hash function to fingerprint a chunk is time-consuming. It is not a severe concern for legacy ferromagnetic hard disks with a low access speed. We conducted a test by doing inline deduplicaton on hard disk, flash memory, and NVM, respectively. A breakdown of execution time indicates that the performance becomes more subject to fingerprinting when the storage device turns to be faster. In particular, the execution time of fingerprinting for hard disk is about 21 percent of all execution time, but rises up to 44 percent for NVM. As a result, a minimization of fingerprinting cost is desired for inline deduplication on the faster NVM.

Based on aforesaid observations, we have proposed a novel deduplication algorithm for NVM, named *NV-Dedup*. The key contributions of NV-Dedup are as follows.

- Leveraging NVM's byte-addressability, we develop a CPU and NVM-favored, fine-grained *metadata table* to manage deduplication metadata. Deduplication metadata for a data chunk, such as fingerprint, reference count and chunk number, are compacted into one table entry that can be loaded into one CPU cache line as an entity. This benefits both performance and consistency.
- We design a lightweight, logless scheme for the consistency of deduplication metadata. In particular, we define and regulate *deduplication transaction* that bundles operations performed to change metadata for newly writing, modifying, or deleting a data chunk.
- We propose a mechanism of *workload-adaptive fingerprinting*for minimal calculation costs. We build an analytical model and a dynamic transition scheme to suit a workload with changing duplication ratio. In brief, the model tells the cost and benefit of a fingerprinting method for a workload with a certain duplication ratio and the transition scheme enables NV-Dedup to transit among different fingerprinting methods by sampling for a workload with dynamic duplication ratio.

We have built a prototype of NV-Dedup in the Persistent Memory File System (PMFS) [10]. Experiments show that, besides saving substantial NVM space, NV-Dedup improves the performance of PMFS by up to 2.1×. Meanwhile, the throughput of NV-Dedup can be up to 3.9× that of state-of-the-art deduplication algorithm.

The remainder of this paper is organized as follows. Section 2 shows the background of deduplication and NVM. Section 3 presents the motivation. Section 4 details NV-Dedup's design. Section 5 shows a prototype for NV-Dedup and evaluation results. Section 6 presents related work. Section 7 concludes the paper.

2 Background

2.1 Inline Deduplication

We choose to do inline deduplication in NVM-oriented file system rather than offline deduplication for two reasons. First, regarding the write endurance of NVM, inline deduplication reduces write operations to NVM, so it benefits the lifetime of NVM. For offline deduplication, write operations must be done in place and deduplicating data is performed afterwards. Second, without writing duplicate data, inline deduplication is supposed to boost file system performance if writing time saved by deduplication is much more than the time for identifying and managing duplicate data.

TABLE 1
Typical Memory Technologies [1], [10], [34], [35]

Memory Technology	Read Latency (ns)	Write Latency (ns)	Write Endurance (*writes per cell*)
DRAM	60	60	10^{18}
PCM	$50 \sim 300$	$150 \sim 1,000$	$10^8 \sim 10^{12}$
STT-RAM	$5 \sim 30$	$10 \sim 100$	10^{15}
NAND Flash Memory	2.5×10^4	$2 \times 10^5 \sim 5 \times 10^5$	$10^4 \sim 10^5$

Unlike offline deduplication [23], [24], inline deduplication works on the fly to eliminate redundant data. The deduplication granularity can be a file or a *chunk*. Deduplicating files is not fit for a general-purpose system as files vary a lot from each other [13], [25]. The size of a chunk can be fixed or variable. Specific chunking methods may be needed to identify the boundary between chunks [26], [27], [28].

In deduplication, a data chunk to be stored is fingerprinted using a hash function, like MD5, SHA-1 or SHA-256. Fingerprinting avoids byte-by-byte comparisons that require high number of CPU cycles [29], [30]. The deduplication searches a new fingerprint among existing fingerprints. A match will save a write by increasing the matched chunk's *reference count* by one; otherwise, the new data will be written with its fingerprint recorded.

Inline deduplication for primary data has attracted much attention [16], [17], [18], [19], [20]. Two major concerns for deduplicating primary data are performance and consistency. State-of-the-art deduplication designs have used different tactics to avoid performance degradations. An NVM buffer is employed by iDedup for I/O staging [16]. Prefetching deduplication metadata is another method that is widely used [17], [19]. An extended version of Dmdedup [19] uses `NODEDUP` hint to avoid deduplicating application data and file system metadata that ask for redundancy or have low duplication ratios. Yet the entire software stack has to be modified for passing and interpreting such hints. Meanwhile, OrderMergeDedup [18] awaits and merges writes to one metadata block into one single write to reduce I/O overheads.

Inline deduplication must ensure that deduplicated data are traceable after a crash. Deduplication metadata must be kept consistent. Along with synchronizations between DRAM and storage device, dedup v1 has a redo log [31] and Dmdedup uses a COW B-Tree [32] to record changes of deduplication metadata. OrderMergeDedup defines a soft updates-style write order [18]. Whereas, logging and COW introduce performance penalties [18], [33].

2.2 NVM and Architectural Supports

The next-generation NVM is a promising medium for computer systems [6], [7], [33], [36], [37], [38], [39], [40]. It differs from mature flash memory in that NVM is byte-addressable and accessed in a shorter latency, as shown in Table 1. PCM has a higher density than DRAM although it has shortcomings at access latency and write endurance. STT-RAM is

advantageous on access latencies and write endurance compared to PCM. Resistive RAM has some similar features to PCM [2], [10]. 3D XPoint comes with a slower access speed, weaker endurance but a higher density than DRAM [4].

Such characteristics enable NVM to be connected onto memory bus, sitting alongside DRAM. In light of consistency, it is non-trivial to store data into NVM. A disk makes an atomic write of a sector (512 B) [8], [41], but modern 64-bit processors natively support an atomic write of 8 B to memory. Using `cmpxchg16b` with `lock` prefix can atomically write 16 B [10], [42]. Data larger than 16 B cannot be atomically written via one instruction as of today, but usually be operated using log or COW for consistency with considerable overhead. Worse, the typical size of CPU cache line is 64 B, and flushing data in multiple cache lines to NVM may not adhere to the order defined in the program [8], [9], [10], [11], [33]. An altered, uncertain writing order is hazardous to consistency. For example, modifying a file's data entails updating the file's last-modified time (`i_mtime`). If modifying data failed due to a crash but updating the file's `i_mtime` completed, the file would seem updated but contain obsolete data.

Reordering writes is transparently done by CPU or memory controller. One method to impose a writing order is using *cache line flush* (e.g., `clflush`, `clflushopt`, or `clwb`) and *memory fence* (e.g., `sfence`) to follow a regular store instruction. `clflush` explicitly invalidates a cache line and flushes the data to NVM. Note that `clflush` is not an atomic write. `sfence` is a barrier which guarantees that store operations issued after the barrier cannot proceed unless those before the barrier have been completed. Thus, using a series of {`sfence`, `clflush`, `sfence`} ensures that multiple cache lines are stored in order. However, the overhead of `clflush` and `sfence` is significant [33].

2.3 NVM-Oriented File Systems

NVM-oriented file systems have emerged for applications to store and access data with NVM [5], [8], [9], [10], [11]. Take PMFS for an example [10]. Both DRAM and NVM sit on the CPU memory bus. The physical space ranges of DRAM and NVM can be specified and separated by configuring the `memmap` option in the GRUB boot loader. On booting up, the NVM space is exposed to be usable for PMFS. PMFS formats NVM space in the unit of 4 KB block for ease of management. The beginning two NVM blocks are two copies of file system `superblock` that redundantly store information of PMFS. The address of the first `superblock` is base address of NVM space. Given an NVM block number, its NVM address can be obtained by left shifting the block number and adding to the base address. After `superblocks` is `PMFS_log` that PMFS uses for consistency of file system metadata like `inodes`. PMFS does not guarantee the consistency of file data. `PMFS_log` is an undo log in implementation. A modified `inode` is first written to `PMFS_log` before updating the `inode` in place. On recovery after a crash, an `inode` that has not been successively updated can be rolled back by using the logged version in `PMFS_log`.

Remainder NVM blocks after `PMFS_log` are used for `inodes` and file data. PMFS makes the `inode table` as a special file with a reserved `inode` number and the file data are `inodes`. PMFS organizes every file as a tree. A file's `inode` holds the tree's root which is a block number referring to an NVM block. The root block is either a data block or an indirect block that stores block numbers of leaf nodes. The leaf nodes of a tree are data blocks of file. In the original design of PMFS there is at most one level of indirect blocks for an `inode`, which enforces a limit on the file size.

PMFS manages NVM blocks via a block allocator which follows OS virtual memory manager for managing main memory pages. PMFS maintains numerous freelists in DRAM. Initially all free blocks are put in one freelist. A block allocation fetches one unused NVM block from the freelist and a block deallocation returns one block to the freelist. PMFS saves allocator structures in a reserved `inode` on a normal unmount. In case of crash, PMFS scans all `inodes` so that NVM blocks that are being occupied by files can be learned and remaining blocks are put into the freelist.

PMFS uses the *eXecute-In-Place* (XIP) to write and read file data, bypassing DRAM page cache [10]. When the underlying storage is byte-addressable NVM, XIP provides a set of callback routines that offer a clean and efficient way to directly access data in NVM without buffering data in DRAM page cache. XIP enables an application to flexibly write and read data of a file at any offset with any size.

In PMFS, writing or reading data of a file is traversing the file's tree. When an application writes data to a new file, an unused `inode` will be allocated from `inode table`. A calculation is done with the quantity of data to be written, so the number of NVM blocks needed is obtained. These NVM blocks will be allocated by block allocator and their numbers are filled in the file's `inode`. Then PMFS writes data to allocated NVM blocks. On the other hand, reading data from a file first retrieves block numbers via the file's `inode`, and then data will be fetched from NVM blocks.

3 Motivation

Deduplicating primary data is profitable for an NVM-oriented file system. First, the valuable NVM space can be saved. Second, considering the limited write endurance of some NVM as indicated in Table 1, fewer write operations are beneficial for the lifetime of NVM device. Furthermore, if an avoidance of writing duplicate data saves more time than the execution time for identifying and managing duplicate data, it may boost the performance of file system.

We have implemented a state-of-the-art deduplication algorithm in PMFS. Deduplication metadata are organized in a block-based COW B-tree [32]. They are buffered in DRAM for use and synchronized to NVM once every 1,000 blocks written by applications [32]. NVM was emulated to be PCM [8], [43], [44]. Fio [45] generated a file and wrote 2048MB data to the file at varied duplication ratios. In Fig. 1, the curve of PMFS is the throughput (IOPS) of original PMFS without deduplication while the other curve is IOPS of PMFS with conventional deduplication (`PMFS-Dedup`).

O1. *Significant write amplification is caused by conventional metadata organization and consistency.* The results in Fig. 1 indicate that the performance is severely impaired by deduplication. For example, the throughput of `PMFS-Dedup` is just 42 percent that of original PMFS at the duplication ratio of 50 percent. We looked into the deduplication algorithm to explore the reason for performance drop. A 4 KB block holds

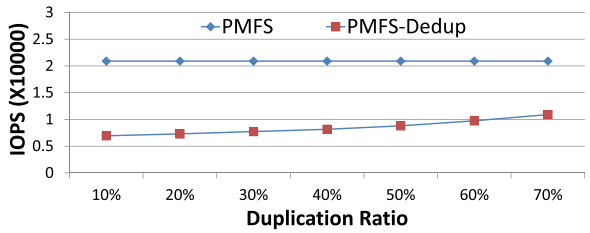


Fig. 1. Performance degradations caused by deduplication.

256 MD5 fingerprints. The block format requires that even an insertion or a removal of one fingerprint should read and write a whole block. Such *write amplification* is adverse to performance. On the other hand, metadata buffered in DRAM have to be synchronized to NVM for consistency. The expected space saved by deduplication is 1,024 MB at a duplication ratio of 50 percent. However, as we observed, by synchronizing metadata once every 1,000 blocks written by applications, 598.4 MB metadata I/Os must be written. Worse, these 1,000 data blocks can be lost upon a sudden crash due to the loss of deduplication metadata.

As NVM is byte-addressable and has DRAM access speed, deduplication metadata can be stored in NVM, which eliminates aforementioned synchronization I/Os. To keep metadata consistent, logging or COW with ordered writes is usually used with substantial performance penalties. In addition, such structures do not take into account the characteristics of deduplication metadata. Therefore, a fine-grained metadata management with light consistency and customization for deduplication metadata is highly required for inline deduplicating data on NVM.

O2. *The performance becomes more subject to fingerprint calculations on the faster NVM.* We also measured the execution time of writing 2,048 MB data at 50 percent duplication ratio with hard disk, flash memory and NVM. The breakdowns of I/O and fingerprinting time are shown in Fig. 2. Fingerprinting a data chunk is non-trivial due to the intricate calculation process of cryptographic hash function. With a slow disk, the time of fingerprinting is inferior and contributes 21.0 percent of running time. Nevertheless, when the NVM is used as storage meduim, the time of fingerprinting rises up to 44.0 percent of running time. Since fingerprinting is on the critical path of deduplicating primary data, reducing the time of fingerprinting is desirable for designing an efficient inline deduplication algorithm on NVM.

4 Design of NV-Dedup

4.1 Overview

The architecture of NV-Dedup is illustrated in Fig. 3. In regard to the access characteristics of deduplication metadata, we develop an in-NVM *metadata table*. It is a fine-grained, CPU and NVM-favored structure, and seeks cache

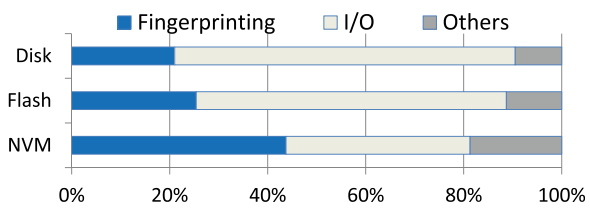


Fig. 2. The cost of fingerprinting with three media.

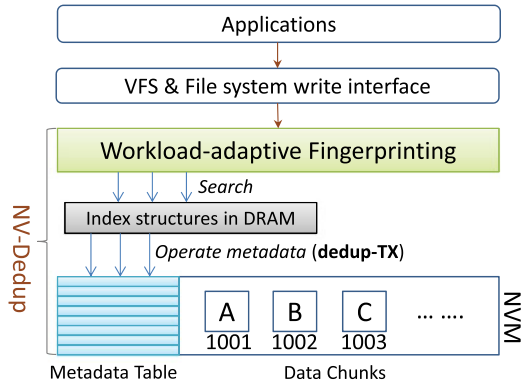


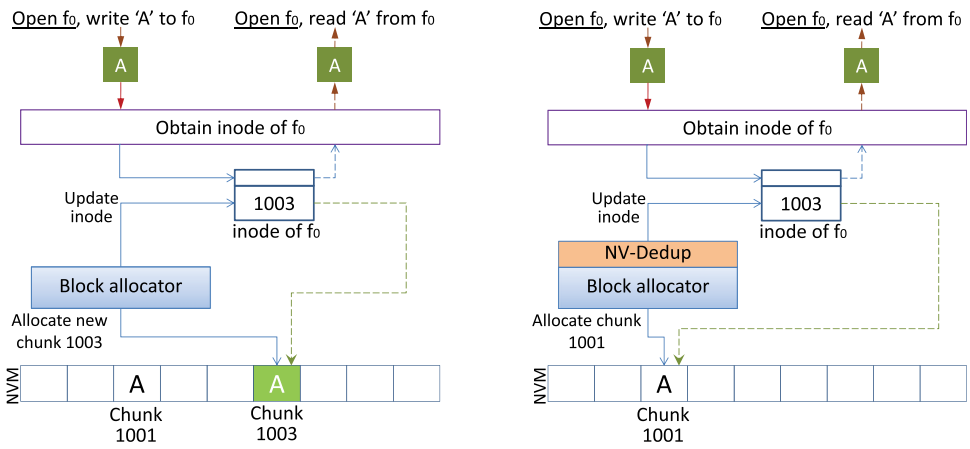
Fig. 3. The architecture of NV-Dedup.

efficiency for performance and atomic write for consistency. We make NV-Dedup directly operate with metadata table and bundle such operations in a lightweight *deduplication transaction* (dedup-TX) for metadata consistency with minimal I/Os. Fingerprinting a chunk is not included in a dedup-TX. In order to reduce the calculation cost of fingerprinting, we design a workload-adaptive fingerprinting mechanism in NV-Dedup. We build an analytical model for NV-Dedup to select a cost-efficient fingerprinting method given a certain duplication ratio. Furthermore, we make NV-Dedup transit among three fingerprinting methods for a workload with dynamic duplication ratio by periodical sampling.

NV-Dedup in File System. NV-Dedup is within an NVM-oriented file system. It makes a deduplication chunk identical to a file system block. As shown in Fig. 3, NV-Dedup first calculates a fingerprint for each chunk, and searches in metadata table via index structures in DRAM to determine if the chunk is duplicate or not. Deduplication in a file system enables NV-Dedup to refrain from the address mapping that incurs both spatial and temporal costs [32], [46], [47]. NV-Dedup cooperates with the aforementioned block allocator of PMFS to deduplicate file data. It does not deduplicate file system metadata, such as `superblock` and `inodes`, as their duplication ratios are very low [19].

NV-Dedup deduplicates data in the write routine of file system and does no change to read routine. Fig. 4 illustrates how file system writes and reads a chunk with data 'A' for file f_0 without and with NV-Dedup. In Fig. 4a, on writing 'A', file system will open f_0, get `inode` of f_0, and allocate a chunk 1003 for 'A' although 'A' has been in NVM chunk 1001. Chunk 1003 is recorded in f_0's `inode`. On reading 'A', the file system opens f_0 again, looks up in `inode` of f_0 for the chunk number 1003, and returns 'A'. At the presence of NV-Dedup, the write routine differs while reading data is not affected. On writing 'A' in Fig. 4b, NV-Dedup indicates that 'A' is found in chunk 1001. So the block allocator will 'allocate' chunk 1001 to update `inode` of f_0 without consuming a free chunk. When reading 'A', chunk 1001 would be accessed by referring to `inode` of f_0.

Chunking of NV-Dedup. A chunk in NV-Dedup is a block with 4 KB size, and must be aligned at block boundary. However, applications can write data with XIP to NVM at any offset with any size. In such scenarios NV-Dedup needs to do chunking over data. The steps are as follows.



(a) Write and read routines in File System (b) Write and read routines with NV-Dedup

Fig. 4. Writing (solid lines) and reading (dash lines) data with and without NV-Dedup in a file system.

(1) According to the offset and quantity of data to be written, NV-Dedup first locates NVM block(s) involved in the write operation.
(2) If an NVM block is to be partially updated, unmodified data in the NVM block will be read out and patched with data to be written to form a complete chunk.
(3) NV-Dedup follows the aforementioned deduplication procedure on the patched chunk.

4.2 Fine-Grained Metadata Management

The in-NVM metadata table is illustrated in Fig. 5. An entry of the metadata table has the same size as a CPU cache line. The first field of an entry has a *tag* (1 B) and an ID of a *deduplication transaction* (7 B). This field is for the use of consistency, which will be detailed in Section 4.3.

The second field is the reference count to indicate the number of references on a data chunk. The third field is the chunk number. Note that each of the three mentioned fields can be updated using an 8 B atomic write. Also, the first and second fields can be jointly updated using a 16 B atomic write (`cmpxchg16b` with `lock` prefix).

The fourth and fifth fields store strong and weak fingerprints, respectively, for a data chunk. 32 B are sufficient for common strong cryptographic hash functions, such as MD5 (16 B), SHA-1 (20 B), and SHA-256 (32 B). 4 B are used for CRC32 or Fletcher-32. The sixth field is a 1 B flag to indicate whether the strong fingerprint is valid or not. Keeping two fingerprints for a chunk is for the use of NV-Dedup's workload-adaptive fingerprinting, as to be shown in Section 4.4. The last 3 B are padding bytes so that an entry has the same size as a CPU cache line.

Tag-TX_ID	Reference Count	Chunk Number	Fingerprint (strong)	Fingerprint (weak)	Strong Fin. Flag	Padding
0x10 1001	1	1001	-	0xE3...27	0	
0x80 2016	1	1002	0x1E0878...26	0x35...45	1	
0x40 3721	3	1003	0x224E5C...33	0xE2...71	1	
...	...	...	...	...	...	
0x20 1275	3	7890	-	0x12...73	1	
8B	8B	8B	32B	4B	1B	3B

Fig. 5. An illustration of NV-Dedup's metadata table.

Two features of the metadata table are as follows.

- An entry of cache line size helps to achieve CPU cache efficiency. We enforce the in-NVM layout of metadata table is cache line-aligned so that an entry can be loaded into one cache line as an entity.
- NV-Dedup jointly stores fingerprints, reference count, and chunk number into one entry for a data chunk, which benefit both performance and consistency. Since metadata of a chunk are likely to be successively used, fitting them into one cache line helps the CPU to load them together and also eases the preservation of consistency with one cache line flush and memory fence.

The spatial cost of metadata table is insignificant. Assuming xGB NVM space with 4 KB per chunk, at most $(\frac{x\text{GB}}{4\text{KB}} \times 64\text{ B})/x\text{GB} \approx 1.6$ percent NVM space will be taken as in the worst case with no duplicate data. We can reserve such contiguous, cache line-aligned NVM space for the metadata table, and utilize a free list to trace unused entries for entry allocations and reclamations.

In deduplication, entries in the metadata table are accessed in two ways: 1) find an entry that contains a fingerprint for writing a data chunk, and 2) find an entry given a chunk number when the chunk is to be deleted. To accelerate such searches, NV-Dedup employs two *index structures* in DRAM: a hash table with fingerprints as search keys and an array indexed by chunk numbers. A lookup of new fingerprint either hits an existing entry, or initiates an index in the hash table to record a newly-created entry for the fingerprint. An element in the array refers to an entry that holds deduplicaton metadata of a chunk.

In implementation, index structures and free list record memory addresses of entries in the metadata table. Given a memory address of 8 B, 24 B (3×8 B) will be needed for each entry (each chunk) by three structures. With 4 KB per chunk in an NVM device of xGB, the spatial cost of three structures in DRAM is about $(\frac{x\text{GB}}{4\text{KB}} \times 24\text{ B}/x\text{ GB}) \approx 0.6$ percent of overall NVM space size. Note that this is the maximum size for three structures because the free list will decrease with more and more data stored.

Index structures and free list are reconstructable by scanning the metadata table after a normal shut-down or a crash.

TABLE 2
Entry Operations Defined for the Metadata Table

Usage Format of Entry Operation	Code	Description
`add_entry`(e, tag, tx_id, fin, chk)	0x10	Add an entry e with chunk number chk, a tag, a fingerprint fin in dedup-TX of ID tx_id
`null_entry`(e, tag, tx_id)	0x80	Nullify an entry e with a tag in dedup-TX of ID tx_id.
`increase_RC`(e, tag, tx_id)	0x20	Increase reference count of an entry e with a tag in dedup-TX of ID tx_id.
`decrease_RC`(e, tag, tx_id)	0x40	Decrease reference count of an entry e with a tag in dedup-TX of ID tx_id.

So they are kept volatile in DRAM. Writing and accessing them with DRAM is much faster than doing so with NVM. We have done experiments on rebuilding index structures and the time spent on rebuilding is insignificant. Details would be shown in Section 5.6.

4.3 Lightweight Consistency Scheme

NV-Dedup is responsible for the consistency of deduplication metadata that determines whether data chunks are still accessible after a crash. Conventional consistency strategies that record metadata changes via COW or logging are workable but inefficient, because additional metadata I/Os are caused. We employ a *lightweight*, logless scheme in NV-Dedup for the consistency of deduplication metadata.

4.3.1 Deduplication Transaction

Inline deduplication must ensure that writing, deleting or modifying *every* chunk is consistently handled. NV-Dedup bundles operations that change metadata for processing *one* chunk into one dedup-TX. A global *TX_ID_Generator* is employed. The process of a dedup-TX is as follows.

1) A dedup-TX starts by copying a unique TX_ID from the TX_ID_Generator.
2) During the dedup-TX, metadata is changed by doing operations with entries of the metadata table.
3) After operated entries are flushed to NVM using cache line flush (`clflush`) and memory fence (`sfence`), the dedup-TX is ended by increasing the TX_ID_Generator by one via an 8 B atomic write.

TX_ID_Generator counts from 1. With 7 B, it hardly wraps around as it can make 2^{56} transactions. A dedup-TX starts by exclusively acquiring TX_ID_Generator. The dedup-TX releases TX_ID_Generator after increasing it in the end. If a crash occurs before the atomic increase of TX_ID_Generator, the dedup-TX is uncompleted and entries involved would have the same TX_ID as TX_ID_Generator. Such inconsistent entries are identified by comparison for a recovery. Note that fingerprinting a chunk is not included in a dedup-TX but starts before it. Fingerprinting a chunk takes much longer time than the duration of a dedup-TX. Fingerprint calculations can be concurrently done for multiple chunks via multi-core computing. Hence the acquisition and release of TX_ID_Generator for a short-lived dedup-TX hardly affects parallel accesses to NVM.

There are three types of dedup-TX, as new write, modification and deletion differ from each other in operating the metadata table.

New Write. To newly write a data chunk, if its fingerprint is not found in the metadata table, a chunk will be allocated to store the data with a new entry added to the metadata table. Thus, an entry operation called `add_entry` is defined. Its usage format is shown in Table 2. An entry is assigned from the free list and all of its fields are cleared to be zeros. `add_entry` writes the data's fingerprint and the allocated chunk number in the entry with a TX_ID copied from the TX_ID_Generator, and sets the reference count to be 1. We give a code of 0x10 to `add_entry` and set it as the entry's tag, so that we can know what has happened to the entry. This is particularly useful in recovery.

If the fingerprint of newly-written data is found in an entry, we will increase the data chunk's reference count. We define another operation called `increase_RC` with a code of 0x20, as shown in Table 2. Leveraging the 16 B atomic write, `increase_RC` forcefully increases the reference count by one together with setting the entry's tag (0x20) and TX_ID. The two actions cannot be separated for consistency. For example, assume a crash that occurs after increasing the reference count but before setting the tag and TX_ID. The inconsistent entry must be recovered as with a greater reference count, but cannot be identified as its TX_ID has not been set with TX_ID_Generator. An inverse order of the two actions are also problematic. They are hence integrated into one 16 B atomic write as the `increase_RC`.

Algorithm 1. Dedup-TX Procedure for a New Write

Input: New data A to be stored for a file f
1: $fin \Leftarrow$ `calculate_fingerprint`(A);
2: $tx\_id \Leftarrow$ TX ID Generator;
3: **if** (A is non-existent data) **then**
4: Chunk chk allocated for A; // tag = 0x10
5: `add_entry` $(e_0, tag, tx\_id, fin, chk)$;
6: **else if** (A is duplicate as entry e_1 is found) **then**
7: `increase_RC` (e_1, tag, tx_id);▷ tag = 0x20
8: **end if**
9: `clflush` e_0 or e_1, `sfence`;
10: Increase the TX_ID_Generator by one with atomic write;
11: Return chunk number to update f's `inode`;

Algorithm 1 sketches the procedure of dedup-TX for a new write using `add_entry` and `increase_RC`. Lines 3 to 5 and 6 to 7 respectively present the aforementioned two cases. Before increasing the TX_ID_Generator to end the dedup-TX (Line 10), the newly-added entry or modified entry will be flushed to NVM by `clflush` and `sfence` (Line 9). A completion of write will be returned with a chunk number to update the file's `inode` (Line 11).

Deletion. There are also two cases to delete a chunk. On one hand, if the reference count is greater than one for the chunk to be deleted, only the reference count needs to be decreased by one. Hence, an operation `decrease_RC` is defined with a code of 0x40. It is also designed to be a 16 B atomic write for consistency like `increase_RC`.

On the other hand, if the reference count is one for the chunk to be deleted, the chunk can be truly deleted and its

corresponding entry will be nullified. We define the fourth entry operation, `null_entry` with a code of 0x80. `null_entry` should clear up an entry. Yet we deem that an entry is nullified and obsolete when its tag is 0x80. `null_entry` writes TX_ID and the tag with the 0x80 code via an 8 B atomic write. The usage formats of `decrease_RC` and `null_entry` are illustrated in Table 2. A nullified entry can be immediately reused in future dedup-TX.

Algorithm 2 shows the procedure of dedup-TX for deleting a chunk. If greater than one, the chunk's reference count will be decreased via `decrease_RC` (Lines 2 to 3); otherwise, the entry will be nullified via `null_entry` (Lines 4 to 5). In either case the modified entry will be flushed to NVM by `clflush` and `sfence` (Line 7) prior to an increase of TX_ID_Generator (Line 8). The file system is then informed of the completed deletion (Line 9).

Algorithm 2. Dedup-TX Procedure for a Deletion

Input: A data chunk with entry e to be deleted for file f
1: $tx\_id \Leftarrow$ TX_ID_Generator;
2: **if** (The data chunk's reference count > 1) **then**
3: `decrease_RC` (e, tag, tx_id); // tag = 0x40
4: **else**
5: `null_entry` (e, tag, tx_id); // tag = 0x80
6: **end if**
7: `clflush` e, `sfence`;
8: Increase the TX_ID_Generator by one with atomic write;
9: Return with the completion of deletion;

Modification. To modify a data chunk is more complicated as two entries will be involved in a dedup-TX, one for the original data and the other one for the updated data. If the reference count of the original data is one while the updated data is non-existent in the file system, a new entry will be added and the entry for the original data should be nullified. We can use `add_entry` and `null_entry` to do so, respectively. This case is overwriting data in a chunk, as illustrated at Lines 3 to 7 in Algorithm 3. The chunk number is not changed for the file.

There are three other cases for a modification. The reference count of original data can be one, but the updated data is duplicate in the file system. Or the reference count of original data is greater than one while the updated data may be either non-existent or duplicate. In such three cases, NV-Dedup needs to update the file's `inode` with a changed chunk number (Line 23). For example, when the reference count of original data is greater than one and the updated data is non-existent in the file system, a chunk should be allocated for the updated data and the original data's reference count would be decreased. In Algorithm 3, `add_entry` and `decrease_RC` are successively conducted (Lines 10 to 11), and the newly-allocated chunk number (chk_1 in Line 9) will be passed to the file's `inode`. The left two cases are illustrated with Line 13 to 20 of Algorithm 3. The file system is adjusted to treat these three cases as new writes with chunk allocations. Before atomically increasing TX_ID_Generator (Line 22), newly-added or modified entries will be flushed to NVM via `clflush` and `sfence` (Line 21).

4.3.2 The Recovery of Deduplication Metadata

File system records a normal shutdown or a crash in its `superblock`. In case of a crash, NV-Dedup is triggered by the file system to scan the metadata table. It leverages TX_ID and tag to identify and recover an inconsistent entry. Entries with zero TX_IDs are treated as free entries. An entry is inconsistent when it has the same TX_ID as TX_ID_Generator. The entry's tag tells what entry operation has been last performed. For `add_entry`, it is revoked by `null_entry`, and vice versa. For `increase_RC`, the entry is rolled back by `decrease_RC`, and vice versa. After NV-Dedup recovers inconsistent entries, TX_ID_Generator is increased by one via an atomic write to end a successful recovery.

Algorithm 3. Dedup-TX Procedure for a Modification

Input: Data A_1 used to modify A_0 in a chunk for file f
1: $fin \Leftarrow$ `calculate_fingerprint`(A_1);
2: $tx\_id \Leftarrow$ TX_ID_Generator;
3: **if** (A_1 is non-existent data) **then**
4: **if** (A_0 with entry e_0's reference count is 1) **then**
5: // A_0 in chunk chk_0, to be overwritten
6: `add_entry` $(e_{1a}, tag_1, tx\_id, fin, chk_0)$; // tag_1 = 0x10
7: `null_entry`(e_0, tag_0, tx_id); // tag_0 = 0x80
8: **else**
9: New chunk chk_1 allocated to store A_1
10: `add_entry`$(e_{1a}, tag_1, tx\_id, fin, chk_1)$; // tag_1 = 0x10
11: `decrease_RC`(e_0, tag_0, tx_id); // tag_0 = 0x40
12: **end if**
13: **else if** (A_1 is duplicate as entry e_{1b} is found) **then**
14: `increase_RC`(e_{1b}, tag_1, tx_id); // tag_1 = 0x20
15: **if** (A_0 with entry e_0's reference count is 1) **then**
16: `null_entry`(e_0, tag_0, tx_id); // tag_0 = 0x80
17: **else**
18: `decrease_RC`(e_0, tag_0, tx_id); // tag_0 = 0x40
19: **end if**
20: **end if**
21: `clflush` involved entries, `sfence`;
22: Increase the TX_ID_Generator by one with atomic write;
23: Return chk_1 or chunk number in e_{1b} to update f's `inode`;

A crash may happen in a recovery. Recovered entries must be prevented from being 'recovered' again. When recovering an entry, NV-Dedup sets an unused bit of the entry's tag as '1' to mark that the entry has been recovered. These bits can be cleared after a successful recovery.

4.4 Workload-Adaptive Fingerprinting

4.4.1 Modeling of Workload-Adaptive Fingerprinting

As indicated in Section 3, the performance becomes more subject to fingerprinting with NVM. The cost of fingerprinting has been a concern even for legacy storage devices. For example, Rsync [48], CAFTL [12] and ZFS [25] utilize a weak and strong fingerprinting method. For a chunk to be stored, a weak fingerprint such as CRC32 is calculated and searched in existing weak fingerprints. Only when a match is found will a strong fingerprint like MD5 be calculated to check if two chunks are truly the same or not. So the strong hash function is occasionally called. The extended Dmdedup [19] yet supports manually passing a hint through software layers to prevent some data and file system metadata from being deduplicated, so that they will not be fingerprinted due to their fairly low duplication ratios.

It is well known that data issued by applications are with different workload characteristics and different duplication ratios [12], [13], [49], [50]. For example, the duplication ratio of

TABLE 3
Notations of Analytical Model

Notations	Descriptions
$T_{nodedup}$	Normal Service Time (without deduplication)
T_{dedup}	Service Time (with deduplication)
D	Number of data chunks
c	Average time to write a data chunk to NVM
s	Average time to calculate strong fingerprint for a chunk
w	Average time to calculate weak fingerprint for a chunk
r	Duplication ratio
λ	Average time to search a fingerprint

data in a home server is quite low (15∼30 percent), and that of a server for file sharing is much higher (40∼50 percent); the duplication ratio can be as high as 90 percent for virtualization libraries [13]. We hence do an intuitive analysis on the *worthiness* of fingerprinting for different workloads.

- Fingerprinting will not be worthwhile if the duplication ratio of a workload is low. Little NVM space can be saved compared to the execution time of fingerprinting.
- A workload with a moderate duplication ratio is considerable for deduplication. A method of weak and strong fingerprinting is feasible for such a workload.
- A workload with a high duplication ratio is favored by deduplication. However, weak fingerprinting may frequently report that a received chunk is duplicate and strong fingerprinting has to be called almost every time for a recheck.

Choosing a fingerprinting method for a certain workload becomes an issue. We have made an analytical model to do so for NVM. Notations to be used are listed in Table 3.

The presence of inline deduplication inevitably affects the performance. If we expect a comparable performance with and without deduplication, we should have

$$T_{dedup} \leq T_{nodedup}. \tag{1}$$

Without deduplication, the time to write D chunks is

$$T_{nodedup} = D \cdot c. \tag{2}$$

With inline deduplication like Dmdedup using a single strong hash function, the service time would be

$$T_{dedup} = D \cdot s + (1 - r) \cdot D \cdot c + \lambda \cdot D. \tag{3}$$

With Equations (1), (2) and (3), we will have

$$r \geq \frac{s + \lambda}{c}. \tag{4}$$

On the other hand, using the method of weak and strong fingerprinting, the service time would be

$$T_{dedup} = D \cdot w + (1 - r) \cdot D \cdot c + r \cdot D \cdot s + \lambda \cdot D. \tag{5}$$

Here we make an estimate for the worst case where all duplicate chunks need to be checked by the strong hash function. With Equations (1), (2), and (5), we can get

$$r \geq \frac{w + \lambda}{c - s}. \tag{6}$$

As s, w, λ and c are measurable, both Equations (4) and (6) can be figured out. They promise two *thresholds* that enable NV-Dedup to utilize different fingerprint methods for different workloads. In brief,

- `Non_Fin` method does not deduplicate data. It is fit for a workload with r that is no higher than $\frac{w+\lambda}{c-s}$. Note that although these chunks are not deduplicated, they are still recorded in the metadata table with a special flag for lacking fingerprints.
- `w_s_Fin` method calculates a weak fingerprint for a data chunk, and uses a strong fingerprint to check data that are not surely identified by comparing weak fingerprinting. It is efficient for a workload with r between $\frac{w+\lambda}{c-s}$ and $\frac{s+\lambda}{c}$.
- `Str_Fin` method calculates a single strong fingerprint for data issued by a workload with r higher than $\frac{s+\lambda}{c}$.

4.4.2 Adaptation to Dynamic Workload

The duplication ratio of workload is changing from time to time, so NV-Dedup should transit among three aforementioned fingerprinting methods. NV-Dedup uses a scheme of *periodical sampling* to online monitor the duplication ratio of workload. In brief, NV-Dedup defines a sampling period as a fixed number of chunks it receives. In these chunks, NV-Dedup counts how many chunks are duplicate. It hence obtains the duplication ratio in that period. NV-Dedup uses such a ratio to decide which fingerprinting method is to be used in the forthcoming period.

Periodical sampling sounds straightforward but can help NV-Dedup timely respond to a dynamic workload. Meanwhile, it does not demand arduous calculations as sampling can be done along with deduplication. However, there are three problems left for NV-Dedup regarding a fluent transition between two different fingerprinting methods.

(1) First, NV-Dedup may be unaware of changed duplication ratio. When NV-Dedup finds the duplication ratio is low, it would follow `Non_Fin` that does not do fingerprinting. Thus NV-Dedup is oblivious of whether the duplication ratio has risen or not although the workload might have changed. In order to solve this problem, NV-Dedup uses `w_s_Fin` to deduplicate data at some time so that it can learn the change of workload's duplication ratio. In implementation NV-Dedup performs `Non_Fin` and `w_s_Fin` alternately in the periods with very low duplication ratio.

(2) The second problem is the missing of weak fingerprints.
   a) Fingerprints of many chunks are not calculated or stored with the `Non_Fin` method. When NV-Dedup switches to deduplicating data, it is uncertain whether a newly-arrived chunk is duplicate or not since some stored chunks lack weak fingerprints. To fill up absent fingerprints, NV-Dedup employs a background thread. It scans the metadata table. When a chunk without weak fingerprint is found, it will calculate and add the chunk's weak fingerprint.

b) With `Str_Fin`, NV-Dedup does not compute the weak fingerprint for a chunk, which also causes an absence of weak fingerprints. Thus, we alter the `Str_Fin` method to calculate both weak and strong fingerprints for a chunk at a high duplication ratio so that no weak fingerprint is missed with `Str_Fin`. When the duplication ratio is high, many data writes can be saved. So the performance gain brought by deduplication is tremendous. By comparison, the calculation of weak fingerprints is much smaller.

(3) The third problem is the missing of strong fingerprints with the `w_s_Fin` method. `w_s_Fin` calculates weak fingerprint on writing every chunk. If the weak fingerprint is not found in the metadata table, NV-Dedup will deem the chunk to be non-existent and the calculation of strong fingerprint needs not be done for the chunk. So the chunk's strong fingerprint is missing in the metadata table. If a newly-arrived chunk has the same weak fingerprint as a stored chunk, NV-Dedup calculates the strong fingerprint of both chunks for further comparison. Then, NV-Dedup updates the entry of the stored chunk by adding the strong fingerprint.

5 Evaluation

5.1 Prototype of NV-Dedup

We have built a prototype of NV-Dedup within PMFS. NV-Dedup can also be prototyped in other file systems designed for NVM. NV-Dedup makes a chunk identical to the default 4 KB block of PMFS. CRC32 and MD5 are default weak and strong hash functions, respectively, for fingerprinting.

In PMFS, writing or modifying file data is done with `pmfs_xip_file_write`, in which fingerprints are calculated and searched in existing fingerprints. Reading file data is unchanged. `pmfs_xip_file_read` fetches data with a block number following the tree of a file. PMFS's block allocation functions, like `pmfs_alloc_blocks`, are adjusted so that a chunk can be 'allocated' again for duplicate data. `pmfs_free_block` is also modified to decide whether a chunk to be deleted can be surely freed or retained with duplicate data. `pmfs_fallocate` that preallocates space is altered to save NVM space.

NV-Dedup reserves a contiguous, cache line-aligned NVM space to store its metadata table, which is initialized or resumed in `pmfs_fill_super`. NV-Dedup also trigers the recovery of deduplication metadata there.

5.2 Evaluation Setup

The machine used for evaluation is equipped with 2×4 GB DRAM and 2×4 GB NVDIMM [33], [51]. As PMFS bypasses DRAM page cache with XIP, the performance is not affected by DRAM size. The NVDIMM space is made visible to PMFS by configuring `memmap` in GRUB boot loader. NVDIMM is a combination of DRAM and NAND flash memory. At runtime NVDIMM is working as DRAM and NAND flash is invisible to the host. On a power failure, NVDIMM stores all the data from DRAM to flash by using supercapacitor to make the data persistent. As this store process is transparent to other parts of the system, NVDIMM is a good alternative of NVM. On the other hand, the CPU in use only supports `clflush`. New instructions for cache line flush, such as `clflushopt` and `clwb`, if available, may further promote cache efficiency of NV-Dedup. The OS is Ubuntu 14.04.1 with Linux kernel 3.11.0 and GCC 4.8.4. In addition, we have configured the write bandwidth of NVM to be 1/8 of DRAM (300 ns read latency and 600 ns write latency) following settings in the latest literatures [8], [43], [44].

We have used two benchmarks that can set the duplication ratio of workload: Fio and Vdbench [52]. In every case with both benchmarks we ran for ten times and the results to be presented are average values. We compared PMFS with NV-Dedup to three competitors: 1) the original PMFS without deduplication (`No_dedup`), 2) an algorithm following Dmdedup [32] implemented in PMFS (`Dedup_block`) that organizes metadata in the block format and synchronizes to NVM once every 1,000 blocks received by PMFS, and 3) a deduplication algorithm (`Dedup_fine_grained`) with a fine-grained COW B-tree in NVM for metadata management. The latter two use strong fingerprinting (MD5 by default) for deduplication.

We did experiments in two ways. In order to verify the advantage of NV-Dedup over other competitors, we first tested in *static* scenarios with fixed duplication ratios. Then we used a dynamic workload to observe how NV-Dedup online responds to workloads with changing duplication ratios. In the static test, we varied the duplication ratio of Fio and Vdbench to be 10, 20, 30, 40, 50, 60, and 70 percent. NV-Dedup learns the duplication ratio at runtime with parameters in Equations (4) and (6) so that it can do workload-adaptive fingerprinting. On our machine, the mean values of s, w, c, and λ for one chunk are 6.2 μs, 0.8 μs, 9.7 μs, and 0.1 μs, respectively. The two thresholds ($\frac{w+\lambda}{c-s}$ and $\frac{s+\lambda}{c}$) mentioned in Section 4.4 are about 25.7 and 64.9 percent, respectively. As a result, NV-Dedup uses the `Non_Fin` method at a 10 and 20 percent duplication ratio. With 70 percent duplication ratio NV-Dedup relies on `Str_Fin` for deduplication. With other duplication ratios NV-Dedup uses the `w_s_Fin` method.

5.3 Performance with Fio

We configured Fio to create a file of 4GB and perform modifications onto the file data until a preset time period had elapsed. As we intend to observe the effect of deduplication at first, the workload was set as random write with single thread. The time period was set to be five minutes in our experiments. The request unit of Fio was 4 KB. The throughput of I/O per second (IOPS) is used as the metric.

Prior to reporting the performance, we have measured the space saved by NV-Dedup. As we ran Fio for five minutes, we recorded the data issued by Fio and data truly stored by NV-Dedup among the 3rd minute. Because of space limitation, we selectively present results at duplication ratios of 10, 30, 50, and 70 percent in Table 4. It is evident that with the latter three duplication ratios, NV-Dedup is able to identify and remove almost all duplicate data. Since NV-Dedup makes a chunk identical to a 4 KB block, it is effective for NV-Dedup to identify and deduplicate data. For the 10 percent duplication ratio with `Non_DEDUP`, NV-Dedup uses periodical sampling by deduplicating in every other period to online

TABLE 4
Data Issued by Fio and Stored by NV-Dedup

Duplication Ratio of Fio	10%	30%	50%	70%
Fingerprinting Method	Non_DEDUP	w_s_DEDUP		Str_DEDUP
Data Issued (MB)	5,141	6,167	7,510	10,752
Data Stored (MB)	4,884	4,317	3,755	3,333
Percentage of Deduplicated Data	5%	30%	50%	69%

monitor duplication ratio. As a result, NV-Dedup deduplicated 5 percent of issued data.

Fig. 6 shows the throughputs of four approaches. Non_dedup yields consistent throughputs with ascending duplication ratios. The throughputs of NV-Dedup, Dedup_fine_grained, and Dedup_block, gradually rise up with ascending duplication ratios. In particular, the throughput of NV-Dedup is $1.0\times - 2.0\times$ that of Non_dedup, respectively, with seven ascending duplication ratios. Thus, NV-Dedup substantially boosts the performance of PMFS by saving writes for duplicate data, especially at a considerable duplication ratio. Meanwhile, NV-Dedup is able to significantly outperform two competitors since the throughput of NV-Dedup is $3.0\times - 3.9\times$ and $1.3\times - 1.4\times$ that of Dedup_block and Dedup_fine_grained, respectively, with seven duplication ratios.

The higher performance of NV-Dedup is partly accredited to NV-Dedup's fine-grained metadata management and consistency. Fig. 7 shows the average amount of metadata flushed to NVM via clflush per request for NV-Dedup, Dedup_fine_grained, and Dedup_block at five duplication ratios. The compact cache line-aligned metadata management enables NV-Dedup to flush the least metadata. In particular, the metadata flushed by NV-Dedup is just 3.2-5.8 percent of metadata flushed by Dedup_block. Also in Fig. 7, metadata flushed per request by Dedup_block decreases as the duplication ratio increases. A higher duplication ratio means less data to be stored with fewer fingerprints. So less deduplication metadata would be flushed by Dedup_block which synchronizes once every 1,000 data blocks written. On the other hand, Dedup_fine_grained uses a fine-grained COW B-tree for metadata. COW B-tree does not update metadata in place. It writes updated metadata to a different location. The tree structure will be adjusted from leaf node to root [32]. A crash that occurs in the process will not cause metadata loss because the old metadata is retrievable. Nonetheless, all the changes to B-tree have to be made persistent into NVM with clflush. Hence COW B-tree incurs performance penalties in deduplicating data, which makes the metadata management

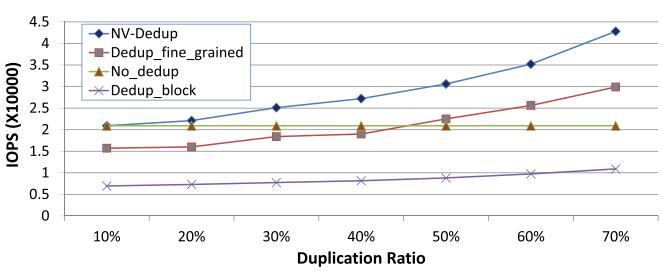


Fig. 6. Throughputs (IOPS) with Fio.

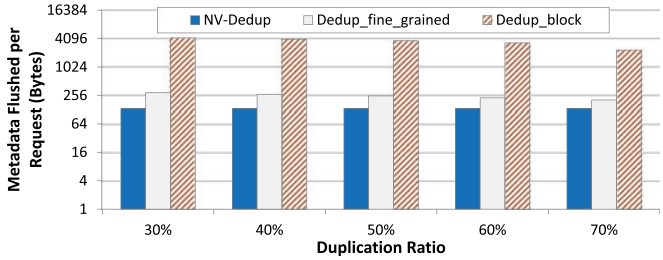


Fig. 7. Metadata flushed per request with Fio.

inferior to NV-Dedup's metadata table for deduplication in NVM. In Fig. 7, Dedup_fine_grained needs to write $2\times$ metadata than NV-Dedup per request.

The higher performance of NV-Dedup is also contributed by its workload-adaptive fingerprinting. NV-Dedup does not deduplicate data or uses the method of weak and strong fingerprinting at low duplication ratios. Dedup_fine_grained and Dedup_block perform strong fingerprinting on chunks. In fact, when the duplication ratio is low, the performance gain by saving writes for duplicate data is not obvious. As illustrated by Fig. 6, NV-Dedup yields much higher throughput than the two competitors at 10-40 percent duplication ratios.

5.4 Performance with Vdbench

We used Vdbench to create a file and write 4 GB data to the file in PMFS with seven duplication ratios. The workload pattern is sequential write in a single thread. The request unit was set to be 4 KB. On the successful writing of all file data, Vdbench would report the throughput of *Requested File Operation Rate*, which is the file write operations per second.

Fig. 8 presents the throughputs for the file generation using Vdbench. The trends of four curves are similar to those in Fig. 6. With seven ascending duplication ratios, the throughput of NV-Dedup is $1.0\times - 2.1\times$ that of Non_dedup, respectively, for sequentially writing data in the file. NV-Dedup outperforms Dedup_block and Dedup_fine_grained by up to $1.3\times$ and $2.0\times$, respectively. The contribution of workload-adaptive fingerprinting is also impressive with Vdbench at low and moderate duplication ratios. With Vdbench, NV-Dedup brings in even more dramatic performance improvements to PMFS, compared to the results with Fio. In particular, NV-Dedup boosts the performance of PMFS by $2.1\times$ at 70 percent duplicate ratio.

We again collected the metadata flushed per request with Vdbench. Fig. 9 show the results. The two competitors still incurred much more metadata I/Os than NV-Dedup. That explains why NV-Dedup could greatly outperform them. However, the metadata flushed per request by NV-Dedup

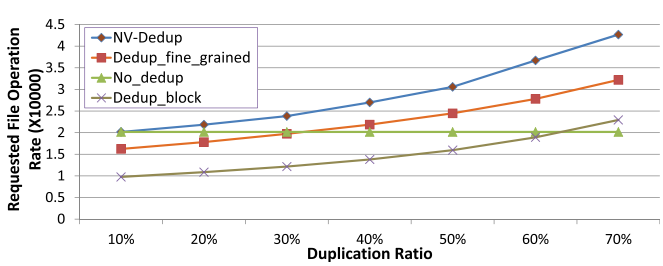


Fig. 8. Throughputs (requested file operation rate) with Vdbench.

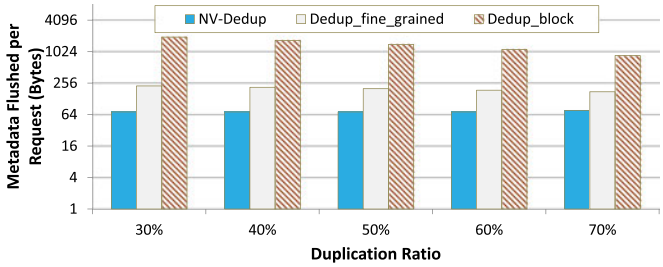
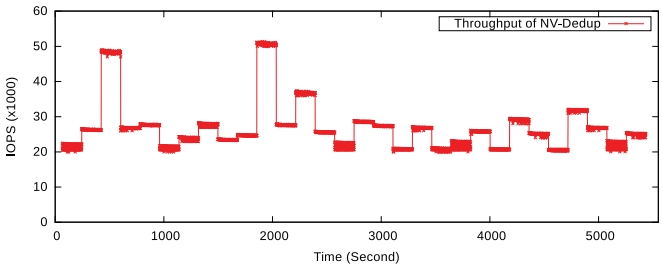


Fig. 9. Metadata flushed per request with Vdbench.

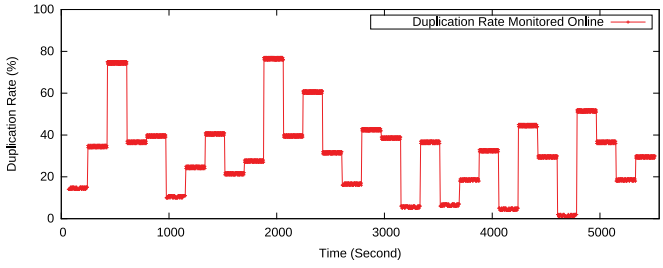
in Fig. 9 become much less than those in Fig. 7. In creating a file with Vdbench, all requests are new writes. As shown in Algorithm 1, only one entry and TX_ID_Generator need to be written for newly writing a chunk. But in the test with Fio, most of the requests would be modifying data of a created file until a preset time period elapsed, and a modification covers two entries and TX_ID_Generator. Metadata I/Os of `Dedup_block` and `Dedup_fine_grained` in Fig. 9 are also much fewer than those in Fig. 7. This drop is still due to the difference between modification and new write, as the latter causes fewer metadata changes.

5.5 Effect of Workload-Adaptive Fingerprinting

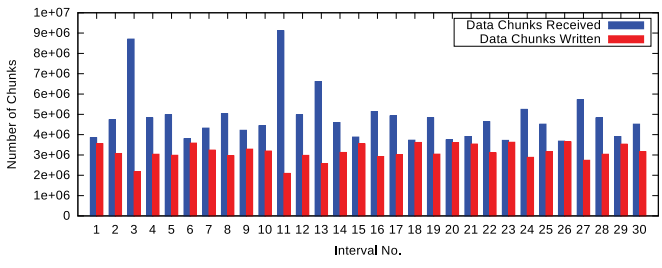
We have done tests to observe the effect of NV-Dedup's workload-adaptive fingerprinting. In implementation, the default length of sampling period is set to be 50,000 chunks received by NV-Dedup. We used Fio to generate a dynamic workload in 30 contiguous intervals with different duplication ratios.



(a) Throughput of NV-Dedup



(b) Duplication Ratio Monitored Online



(c) Number of Chunks Written by NV-Dedup

Fig. 10. The effect of NV-Dedup's workload-adaptive deduplication.

TABLE 5
Recovery Time of NV-Dedup

Duplication Ratio	30%	50%	60%	70%
Entries Scanned (×1000)	734.0	524.3	414.2	325.1
Recovery Time (ms)	220.2	157.3	124.3	97.5

Each interval lasted for 180 seconds with a duplication ratio that was randomly picked. In particular, on randomizing a duplication ratio in an interval, we have considered the possibility of duplication ratio in real environment. In the most intervals the workload is with a low or medium ($\leq$50 percent) duplication ratio. We recorded IOPS of NV-Dedup, the duplication ratio online monitored by NV-Dedup, and the number of chunks issued and written by NV-Dedup. These results are shown in Fig. 10.

The curve in Fig. 10a confirms NV-Dedup's capability of online reaction to a dynamic workload as the throughput is fluctuating to suit the changing duplication ratio. The workload-adaptive deduplication of NV-Dedup is enabled by the online dection of duplication ratio. Once the duplication ratio is detected with NV-Dedup's periodical sampling, NV-Dedup will select a method for deduplication. Because of the memory access speed and frequent sampling, NV-Dedup is able to swiftly adapt to changed duplication ratio. Fig. 10b describes the duplication ratio detected with periodical sampling while Fig. 10c shows the number of chunks practically written by NV-Dedup after deduplication. The two diagrams complement each other and further indicate the deduplication effectiveness of NV-Dedup.

5.6 Recovery Test of NV-Dedup

We ran Fio for five minutes on a 4 GB file with four duplication ratios. Given a duplication ratio, we randomly powered off the machine for ten times. At startup NV-Dedup scans the in-NVDIMM metadata table for recovery and rebuilding index structures. Table 5 shows the *average* values of ten times, i.e., the average number of entries scanned and the average time at a duplication ratio. For all ratios, the time needed to recover deduplication metadata is insignificant. For example, at 50 percent duplication ratio, it was just 157.3 ms by scanning about 524,300 entires.

5.7 Discussion

5.7.1 The Impact of Multi-Threading

We have done experiments to verify the ability of NV-Dedup to support multi-threading access. We used Fio to generate multi-threading workload that concurrently writes data to 1,000 files with 1 MB per file. We configured the duplication ratio to be 30 and 50 percent and varied the number of threads to be 1, 2, 4, 6, and 8. Fig. 11 shows the throughputs in IOPS. It is evident that the throughput rises up with the increase of threads. In particular, with both duplication ratios, the throughput doubles when the number of threads doubles.

These results confirm that the design of NV-Dedup is suited to multi-threading access. Let us do a qualitative analysis. Each thread of Fio goes through NV-Dedup to write data chunks to a file. NV-Dedup first calculates fingerprint for a chunk. Fingerprinting is not included in a dedup-TX. It

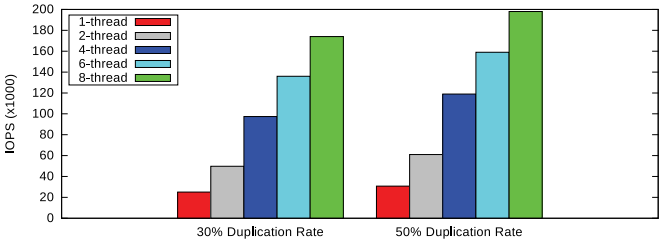


Fig. 11. The impact of multi-threading on NV-Dedup.

then follows Algorithm 3 to operate the metadata table with a TX_ID. Note that fingerprinting is the most time-consuming procedure in deduplication and costs much longer time than operating with one or two entries in the metadata table. When multiple threads write data chunks to files, NV-Dedup can calculate their fingerprints in parallel. Although a thread exclusively operates the metadata table with a TX_ID at a time, such a duration is short and incurs marginal impact to performance. Thus, NV-Dedup is able to effectively support multi-threading concurrent access.

5.7.2 The Impact of NVM Latencies

The default write latency of NVM is 600 ns in above experiments. We also configured the write latency as 400, 500 and 700 ns to see the impact of different NVM technologies. We ran Fio at 50 percent duplication ratio and show results in Fig. 12. In general, performance of NV-Dedup decreases with slower NVM technology.

As shown in Fig. 12, the performance gap between NV-Dedup and `No_dedup` grows from 1.4× to 1.6× as the write latency of NVM varies from 400 to 700 ns. It is because NV-Dedup saves more time by avoiding duplicate writes for slower NVM. Note that the performance gap between NV-Dedup and `Dedup_block` keeps consistent. Yet the difference gap between NV-Dedup and `Dedup_fine_grained` narrows from 1.5× to 1.3× with four latencies. Although both of them are fine-grained in metadata management, NV-Dedup is more efficient in metadata consistency. As a result, NV-Dedup is more sensitive to a longer write latency.

5.7.3 The Impact of Hash Functions

We measured the impact of hash function by running Fio at a duplication ratio of 50 percent. MD5, SHA-1, and SHA-256 were used. MD5 and SHA-1 have similar calculation costs, while SHA-256 is more heavyweight. The throughputs of three hash functions are shown in Fig. 13. NV-Dedup suffers the most from the shift from MD5 to SHA-256. Because NV-Dedup manages metadata in a lightweight way, the cost of fingerprinting would have more evident

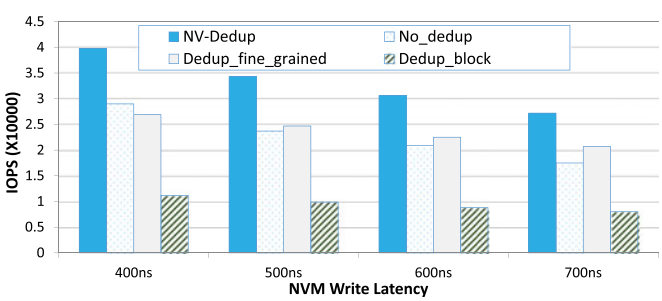


Fig. 12. Fio throughputs on three NVM latencies.

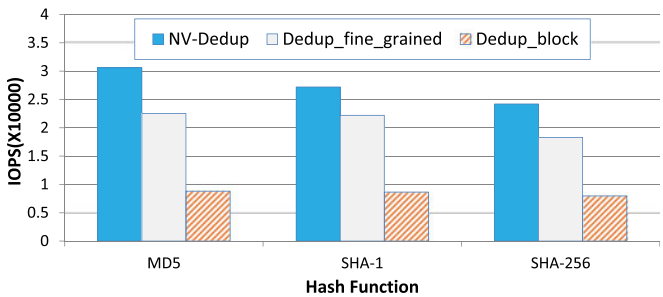


Fig. 13. Fio throughputs on three hash functions.

impacts. This also confirms that the performance will become more subject to fingerprinting for NVM if deduplication metadata are efficiently managed.

6 Related Work

NVM-oriented file systems have been developed to exploit the performance potential of NVM. BPFS [5] is one of the first file systems for NVM. It thoroughly outperforms a traditional file system mounted on a RAM disk. SCMFS [9] tries to minimize CPU overheads for file operations and keep a contiguous space for a file to simplify write/read requests. NOVA [8] applies a log-structured scheme to leverage the fast random access of NVM, and also maintains a log for each file to improve concurrency. HiNFS [11] has observed the slow write of NVM and manages a write buffer in DRAM to accelerate the performance.

The performance is also important for deduplication [13], [20], [53], [54], [55], [56]. The tactics to minimize performance overheads used by previous work, like Dmdedup [19], [32], CAFTL [12], and OrderMergeDedup [18], have been addressed.

iDedup [16] utilizes an NVM buffer for I/O staging. It makes a compromise between performance and deduplication for on-disk storage. Sequential access is favored by disks, but deduplication may break a sequential access as some chunks have been deduplicated. iDedup does deduplication only when duplicate data form a long sequence on disk. Such data locality was also explored by other designs [17], [23], [57]. Zhu et al. [57] proposed to store fingerprints in the same order as their corresponding data chunks on disk. These fingerprints are loaded together as they may be successively accessed. A Bloom Filter is also used in [57] to quickly test whether a data chunk is nonexistent or not.

Data similarity has been considered in deduplication as well [29], [49], [54]. In particular, SiLo [29] counts on both similarity and locality of data to reduce performance overheads. It manages data in the format of a large segment. Correlated small files are bundled into a segement and a large file is partitioned into segments. SiLo also exploits data locality by grouping contiguous segments into blocks to complement its similarity detection.

Metadata consistency is another concern in deduplication. Dmdedup [32], OrderMergeDedup [18] and dedup v1 [31] have been discussed. Ng et al. [46] proposed to treat deduplication metadata as a part of file system metadata. They modified the Journaling Block Device (JBD) with Ext4 to record changes of deduplication metadata. In the design

of El-Shimi et al. [13], when deduplicating data of a file, the system transactionally replaces the file with a virtual file stub that refers to stored chunks.

Flash memory has been involved in deduplication [12], [18], [20], [26], [47], [54], [58] because of its non-volatility and higher throughput than hard disk. ChunkStash [26] accelerates deduplication by using flash to store deduplication metadata in key-value pairs: a fingerprint is a key and the information (size and location) of a chunk is the value. Deduplication in a flash device has also attracted wide attention [12], [47], [54], [58]. Recently Li et al. [20] proposed to integrate deduplication with flash caching.

Main memory deduplication [59], [60] has also been studied to reduce memory footprints of processes by merging equal pages to be one copy. It differs a lot from deduplication for persistent storage. Pages are allocated first. A scan is periodically activated. Two pages, if found to be identical via `memcmp_page`, will be merged. Metadata for merging are not kept consistent due to the volatility nature of DRAM-based main memory.

7 Conclusion

In this paper we propose NV-Dedup, an inline deduplication algorithm to help NVM-oriented file systems deduplicate data while achieving high performance. NV-Dedup maintains a CPU and NVM-favored metadata table, and uses a lightweight scheme for metadata consistency. With an analytical model and periodical sampling, NV-Dedup does workload-adaptive fingerprinting to minimize calculation overheads. A prototype of NV-Dedup has been built in PMFS. Evaluation results show that NV-Dedup effectively saves NVM space by deduplication. Furthermore, it substantially boosts the performance by as much as 2.1×.

Acknowledgments

This work was done when Chundong Wang was with Data Storage Institute, A*STAR.

References

[1] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 2–13.

[2] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, "Understanding the trade-offs in multi-level cell ReRAM memory design," in *Proc. 50th Annu. Des. Autom. Conf.*, 2013, pp. 108:1–108:6.

[3] Z. Sun, et al., "Multi retention level sTT-RAM cache designs with a dynamic refresh scheme," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 329–338.

[4] Micron and Intel, "3D XPoint technology." 2015. [Online]. Available: http://www.micron.com/about/innovations/3d-xpoint-technology

[5] J. Condit, et al., "Better I/O through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 133–146.

[6] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 61–75.

[7] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 1–14.

[8] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol.*, Feb. 2016, pp. 323–338.

[9] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 39:1–39:11.

[10] S. R. Dulloor, et al., "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 15:1–15:15.

[11] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 12:1–12:16.

[12] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 77–90.

[13] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication—Large scale study and system design," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 285–296.

[14] B. Mao, H. Jiang, S. Wu, and L. Tian, "Leveraging data deduplication to improve the performance of primary storage systems in the cloud," *IEEE Trans. Comput.*, vol. 65, no. 6, pp. 1775–1788, Jun. 2016.

[15] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 14–23.

[16] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 24–24.

[17] A. Wildani, E. Miller, and O. Rodeh, "HANDS: A heuristically arranged non-backup in-line deduplication system," in *Proc. IEEE 29th Int. Conf. Data Eng.*, Apr. 2013, pp. 446–457.

[18] Z. Chen and K. Shen, "OrderMergeDedup: Efficient, failure-consistent deduplication on flash," in *Proc. 14th USENIX Conf. File Storage Technol.*, Feb. 2016, pp. 291–299.

[19] S. Mandal, et al., "Using hints to improve inline block-layer deduplication," in *Proc. 14th USENIX Conf. File Storage Technol.*, Feb. 2016, pp. 315–322.

[20] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "CacheDedup: In-line deduplication for flash caching," in *Proc. 14th USENIX Conf. File Storage Technol.*, Feb. 2016, pp. 301–314.

[21] R. Rivest, "The MD5 message-digest algorithm," RFC Editor, Apr. 1992, https://tools.ietf.org/html/rfc1321

[22] FIPS 180–4, "Secure hash standard (SHS)." Aug. 2015. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180–4/fips-180-4.pdf

[23] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 111–123.

[24] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 1–14.

[25] J. Bonwick, "ZFS deduplication." 2015. [Online]. Available: https://blogs.oracle.com/bonwick/entry/zfs_dedup

[26] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up inline storage deduplication using flash memory," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2010, pp. 215–229.

[27] J. Li, et al., "Secure distributed deduplication systems with improved reliability," *IEEE Trans. Comput.*, vol. 64, no. 12, pp. 3569–3579, Dec. 2015.

[28] W. Xia, et al., "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2016, pp. 101–114.

[29] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Similarity and locality based indexing for high performance data deduplication," *IEEE Trans. Comput.*, vol. 64, no. 4, pp. 1162–1176, Apr. 2015.

[30] W. Xia, H. Jiang, D. Feng, and L. Tian, "DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads," *IEEE Trans. Comput.*, vol. 65, no. 6, pp. 1692–1705, Jun. 2016.

[31] D. Meister and A. Brinkmann, "dedupv1: Improving deduplication throughput using solid state drives (SSD)," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–6.

[32] V. Tarasov, et al., "Dmdedup: Device mapper target for data deduplication," in *Proc. Ottawa Linux Symp.*, Jul. 2014, pp. 83–95.

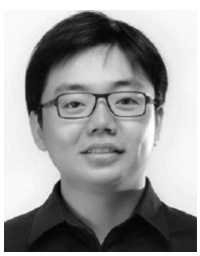
[33] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol.*, Feb. 2015, pp. 167–181.

[34] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 91–104.
[35] R. Chen, Y. Wang, D. Liu, Z. Shao, and S. Jiang, "Heating dispersal for self-healing NAND flash memory," *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 361–367, Feb. 2017.
[36] J. Coburn, et al., "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 105–118.
[37] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 73–80.
[38] J.-Y. Jung and S. Cho, "Memorage: Emerging persistent RAM based malleable main memory and storage architecture," in *Proc. 27th Int. ACM Conf. Supercomput.*, 2013, pp. 115–126.
[39] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 14:1–14:14.
[40] E. H. M. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "Designing an efficient persistent in-memory file system," in *Proc. IEEE Non-Volatile Memory Syst. Appl. Symp.*, Aug. 2015, pp. 1–6.
[41] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *Proc. USENIX Conf. File Storage Technol.*, 2012, pp. 101–116.
[42] Intel, "Intel® 64 and IA-32 architectures software developer manuals," vol. 2, Chapter 3.2, Jun. 2015.
[43] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 385–395.
[44] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *Proc. 20th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 3–18.
[45] J. Axboe, "Fio." Aug. 2016. [Online]. Available: https://github.com/axboe/fio
[46] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. C. Lee, and J. C. S. Lui, "Live deduplication storage of virtual machine images in an open-source cloud," in *Proc. 12th ACM/IFIP/USENIX Int. Conf. Middleware*, 2011, pp. 81–100.
[47] H. Park, S. Kim, O. Bae, K. Kim, J. Kim, and S. Shin, "Nonvolatile memory device and related deduplication method," U.S. Patent 20 140 281 361 A1, Sep. 18, 2014.
[48] A. Tridgell, *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian Nat. Univ., Canberra, ACT, Australia, Feb. 1999.
[49] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve I/O performance," in *Proc. 8th USENIX Conf. File Storage Technol.*, 2010, pp. 1–14.
[50] D. Harnik, E. Khaitzin, and D. Sotnikov, "Estimating unseen deduplication—from theory to practice," in *Proc. 14th USENIX Conf. File Storage Technol.*, Feb. 2016, pp. 277–290.
[51] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 401–410.
[52] H. Vandenbergh, "Vdbench." 2016. [Online]. Available: http://www.oracle.com/technetwork/server-storage/vdbench-downloads-1901681.html
[53] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 1–14.
[54] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, pp. 91–104.
[55] M. Fu, et al., "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2014, pp. 181–192.
[56] M. Fu, et al., "Design tradeoffs for data deduplication performance in backup workloads," in *Proc. 13th USENIX Conf. File Storage Technol.*, Feb. 2015, pp. 331–344.
[57] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 18:1–18:14.
[58] J. Kim, et al., "Deduplication in SSDs: Model and quantitative analysis," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.
[59] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proc. Linux Symp.*, Jul. 2009, pp. 19–28.
[60] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: More effective memory deduplication scanners through cross-layer hints," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 279–290.

Chundong Wang received the bachelor's degree in computer science from Xi'an Jiaotong University, in 2008, and the PhD degree in computer science from National University of Singapore, in 2013. Currently, he is a postdoctoral research fellow in Singapore University of Technology and Design. His research interests include file system, data deduplication, and next-generation non-volatile memory.

Qingsong Wei received the PhD degree in computer science from the University of Electronic Science and Technologies of China, in 2004. He was with Tongji University as an assistant professor from 2004 to 2005. He is currently with the Data Storage Institute, A*STAR, Singapore, as a senior research scientist. His research interests include non-volatile memory system, memory-centric computing, distributed file and storage system, cloud computing, and operating system. He is a senior member of the IEEE.

Jun Yang received the bachelor's degree in computer science from Shanghai Jiaotong University, in 2007, and the PhD degree in computer science and engineering from Hong Kong University of Science and Technology, in 2013. He is currently a scientist in the Data Storage Institute, A*STAR, Singapore. His research interests include database systems, file systems, and next-generation nonvolatile memory.



Cheng Chen is a postgraduate student for PhD degree in National University of Singapore. He received the Bachelor's degree in computer science from Chengdu University of Information Technology (2004-2008), and the Master's degree in computer science from the University of Electronic Science and Technology of China (2009-2012). He is a research engineer in Data Storage Institute, A*STAR, Singapore. His research interests include nonvolatile memory-based operating system and database systems.

Yechao Yang received the bachelor's degree in mechanical and electronic engineering from the University of Science and Technology of China, in 1998. He is a research engineer in the Data Storage Institute, A*STAR, Singapore. Before, he joined DSI, he was a R&D engineer in Hewlett-Packard (Singapore). His research interests include non-volatile memory-based storage, file system, and database systems.

Mingdi Xue received the bachelor's degree in information and telecommunication engineering from Xi'an Jiaotong University, in 2001 and the master's degrees from Xi'an Jiaotong University, in 2004, and National University of Singapore, in 2011, respectively. He is a senior research engineer in the Data Storage Institute, A*STAR, Singapore. His research interests include non-volatile memory-based storage, file system, and database systems.