I/O Causality Based In-line Data Deduplication for Non-Volatile Memory Enabled Storage Systems

Haikun Liu, *Member, IEEE,* Xiaozhong Jin, Chencheng Ye, *Member, IEEE,* Xiaofei Liao, *Member, IEEE,* Hai Jin, *Fellow, IEEE,* Yu Zhang, *Member, IEEE*

Abstract—Data deduplication technologies are widely exploited to reduce capacity demands for storage. Previous chunk-based offline deduplication technologies often cause serious performance overhead due to data chunking and indexing. Particularly, they are not efficient for *non-volatile memory* (NVM) based storage systems because they cannot fully exploit the byte-addressability feature of NVMs for fine-grained deduplication. In this paper, we propose *I/O Causality based In-line Deduplication* (ICID) to maximize the deduplication ratio for NVM-based storage systems. Unlike previous inline deduplication schemes that use hash indexes to identify duplicate data slices, ICID records memory-copy operations in a B-tree structure to achieve causality-based inline deduplication. We propose two novel techniques to manage memory-copy records in the B-tree efficiently. First, to speed up the B-tree lookup, we group memory-copy records targeted to the same page in a B-tree node to improve data locality. Second, we exploit the spatial locality of memory accesses to identify outdated memory-copy records, and delete them in time to reduce memory consumption of the B-tree. We evaluate ICID in a system equipped with Intel Optane DC Persistent Memory Modules. For a typical KV store–LevelDB, our experimental results show that ICID achieves up to 16× higher deduplication ratio and reduces the time cost of data deduplication by 47% on average compared with state-of-the-art deduplication schemes.

Index Terms—Data Deduplication, I/O Causality, Non-Volatile Memory.

1 Introduction

Data deduplication techniques have long been studied for space saving in storage systems. They eliminate duplicate data at file or chunk levels by identifying the same data using cryptographic hash functions such as SHA-1 or MD5. Data deduplication can be performed in an offline or inline mode [1]. The offline deduplication is conducted after the data has been written to the storage device, and thus usually causes write amplification. In contrast, the inline deduplication detects the duplicate data before writing it back into storage, and thus can reduce data redundancy and the wear of storage devices. More importantly, inline deduplication has the potential to explore the correlation among I/O operations at runtime, and may significantly improve the deduplication ratio.

Non-volatile memory (NVM) technologies such as Intel 3D Xpoint [2] offer much lower cost, higher density and energy efficiency than traditional DRAM technologies [3], [4]. They have become a promising complement to DRAM to bridge the performance gap between main memory and SSD/HDD storage devices [5]. However, NVMs often suffer from limited write endurance, a typical PCM cell can only sustain 10^7-10^8 writes [4], [6]. Inline deduplication is a promising approach for reducing the storage consumption and the wear of NVM devices. Unfortunately, existing inline deduplication schemes are not efficient for new NVM devices. Since previous data deduplication techniques [7], [8] are designed for block devices such as HDD and SSD, they suffer from high computation and storage overhead due to data chunking and indexing. For example, to chunk 1 TB data without any redundancy using 4 KB blocks, *Content-Defined Chunking* (CDC) [9] algorithm has to compute rolling hashes for almost 10^{12} times during the chunking stage. Moreover, the SHA-1 algorithm introduces 5 GB fingerprints (20 bytes per fingerprint) to index those chunks [10], [11]. Since DRAM resource is expensive and limited, many recent proposals store most fingerprints on disk. Thus, it is expensive to search those fingerprints in storage [11], [12].

Moreover, existing inline deduplication schemes have to make a tradeoff between the deduplication ratio and the read throughput. Since the performance difference between the sequential read and the random read is significant for SSD/HDD devices, as shown in Table 1, using big chunks (such as 4 KB or 8 KB) [13], [14] can mitigate the performance degradation due to random accesses to SSD/HDD, and also avoids read amplification if the chunk is larger than the disk block. Moreover, using big chunks can also mitigate the total size of fingerprints and the cost of indexing. In contrast, using small chunks can improve the deduplication ratio, but lowers the read throughput and increase the cost of indexing. Currently, most deduplication systems tend to use big chunks to improve the read throughput and to mitigate the cost of indexing, at the expense of a lower deduplication ratio.

The advent of byte-addressable NVM devices such as Intel Optane *DC Persistent Memory Modules* (DCPMM) [2] offers an opportunity to use small chunks for data deduplication because the random read throughput of the NVM device is much higher than that of SSD/HDD devices. As shown in Table 1, the random read throughput of NVM is as high as 74% of its sequential read throughput. Its random/sequential ratio of read throughput is about 9∼20× higher than that of SSDs, and 104× higher than that of HDD. In-line deduplication systems using small chunks can

• *All authors are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China.*
E-mail: {hkliu, xzjin, yecc, xfliao, hjin, zhyu}@hust.edu.cn

TABLE 1: The performance characteristics of different storage devices including Intel Optane NVM, three SSDs, and HDD. The models of three SSDs are WD Blue SN550 NVMe, HP EX900 NVMe, and WD Blue SATA.

Devices	Read Throughput (MB/s)		
	Sequential	Random	Rand./Seq.
NVM	1256	936	73.99%
SSD_1	1240	45	3.63%
SSD_2	998	38.7	3.88%
SSD_3	479	38	8.10%
HDD	161	1.15	0.71%

achieve high deduplication ratio, and thus extend NVMs' lifetime by reducing the amount of data written to NVMs. However, with smaller chunks, existing chunk-based deduplication systems still suffer significant performance and storage overhead [10], [11] because 1) more chunks are generated, and more fingerprints should be calculated; 2) more fingerprints should be stored in memory and are compared during data deduplication.

To achieve fine-grained data deduplication with low cost for these in-memory file operations, we design an *I/O causality-based inline deduplication* (ICID) scheme by fully exploiting the byte-addressability of NVMs. Unlike previous inline deduplication schemes that identify duplicate data by fingerprints at the granularity of chunks (4 or 8 KB), ICID takes the advantage of byte-addressable NVMs to achieve inline deduplication at the granularity of bytes. ICID provides a set of new APIs to record the mapping between the source address and the destination address of each memory-copy operation in a B-tree structure. According to the I/O causality between data copying and file writing operations, ICID is able to find the duplicate data in fine-grained sizes.

ICID achieves data deduplication in three steps. First, when ICID opens a file, it maps the file to the main memory. Second, when the file is read, ICID converts the read operation into a memory-copy operation. ICID records the memory-copy operation in a B-tree (called "rec-tree" in the remainder of this paper) for each process. Third, when the updated data in the DRAM is written back to the storage, ICID accelerates data comparation between the written data and the original data using *Advanced Vector Extensions* (AVX) hardware, i.e., SIMD instructions, and finds out the actual duplicate data slices. Unlike traditional chunk-based deduplication schemes that identify duplicate data without application semantics, ICID can exploit the I/O causality in a program to minimize the search space of data redundancy.

However, it is usually expensive to maintain all memory-copy operations for I/O-intensive applications. ICID still faces two major challenges to manage memory-copy records efficiently. First, how to organize the records of memory-copy operations with low cost of lookup? Second, how to identify and delete outdated records in the rec-tree effectively? To address these challenges, we propose two key technologies as follows:

- **Using a hybrid data structure to store memory-copy records.** To achieve low-latency lookup, we organize memory-copy records in a B-tree. Each node in the tree contains all records of memory-copy operations targeted to the same memory page, and is indexed by the page number. In each node, records are organized in a linked-list/array structure. In this way, we manage records at the page granularity to decrease the number of nodes, and thus reduces the latency of the B-tree lookup. Moreover, since the access pattern of most records features spatial locality, storing adjacent records in a continuous memory space accelerates the data retrieval.
- **Garbage collection for outdated memory-copy records.** To find out useful memory-copy records in the rec-tree quickly, ICID exploits the order between adjacent memory-copy operations within a single page to delete outdated records. If the destination address of a newly-inserted record is lower than the highest one in the current page, ICID deletes the record whose destination address is higher than the inserted one's because the memory of high address will most likely be updated in the future, and thus a new record would replace the old ones.

We implement ICID as a stand-alone library using about 3500 lines of code (LOC). In addition, we develop a user-space filesystem to support ICID based on FUSE [15]. For a typical KV store–LevelDB, our evaluation shows that ICID achieves up to 16× higher deduplication ratio and reduces the cost of data deduplication by 47%, compared with state-of-the-art deduplication schemes.

2 Background and Related Work

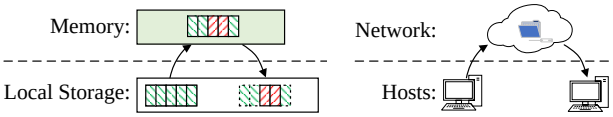
In this section, we first introduce some necessary background, and then describe the related work.

2.1 Chunk-based Data Deduplication

Chunk-based data deduplication is usually implemented as follows. 1) **Chunking and Hashing**: the deduplication system splits the data into different chunks and calculates the fingerprint (hash value) of each chunk. 2) **Indexing and Deduplication**: the deduplication system uses the fingerprint to find a redundant chunk and only keeps the unique one. Specifically, for each chunk, the deduplication system searches its fingerprint from a set of fingerprints. If the fingerprint already exists, the chunk is identified as a redundant one and should be abandoned. Otherwise, the chunk is stored in the storage system, and its fingerprint is added into the fingerprint set.

2.2 Data Redundancy Characteristics

In real-world scenarios, there have been a wide range of file processing applications that need to frequently edit (update) existing files, such as office software, video editors, and so on. Typically, these applications manage files in three steps: 1) read data from disk, 2) modify the data in main memory, and 3) write the modified data back to storage in a new place. Because the modified data is not written in an in-place manner, there are often a high degree of redundancy between the original data and the modified data. In contrast, ICID tracks these I/O operations to identify redundancy and only write the modified portion to storage, thereby saving storage space.



(a) Applicable scenarios (b) Inapplicable scenarios

Fig. 1: Applicable and inapplicable scenarios for ICID.

Many previous studies have observed the spatial locality of data redundancy [16], [17], [18], i.e., most redundant data are physically continuous. For such a large continuous redundant data, traditional chunk-based redundancy detection approaches usually have to split it into many chunks and calculate their hashes one by one, suffering high computation overhead due to chunking and hashing. The spatial locality characteristic motivates us to explore I/O semantic for identifying continuous redundancy, thereby reducing the search space of redundancy detection.

We note that this work focuses on data deduplication for in-memory file operations. As illustrated in Figure 1(a), ICID only deduplicates data that is modified from local storage. In other scenarios where redundant data originates from remote sources, as shown in Figure 1(b), there is no I/O causality. Moreover, ICID also cannot be used for scenarios such as compression/decompression, encryption/decryption, and pre-existing redundancy in a file. In these cases, chunk-based data deduplication approaches [9], [18], [19], [20] are more appropriate.

2.3 Related Work

Deduplication can be divided into file-level and chunk-level according to the data granularity. Chunk-based deduplication is more popular because it identifies redundancy at a smaller granularity, and thus can achieve a higher deduplication ratio [10]. To split a file, the deduplication system needs to find an appropriate cutting point for all chunks and calculate their fingerprints. The file may be split by Fixed-Size Chunking (FSC) [19] or more complex Content-Defined Chunking (CDC) [9]. FSC splits a file into fixed-size chunks, and offers extremely low cost of chunking. However, it usually suffers from a boundary-shift problem [10], i.e., an insertion/deletion in the input stream would result in continuous boundary-shifting for all chunks after the updated position, resulting in a significant decline of the deduplication ratio. CDC addresses this problem by calculating the cutting point according to the data content, and thus the cost of chunking is much higher than that of FSC. As shown in Figure 2, CDC continuously calculates the fingerprint (*fp*) of the sliding window. If the hash value satisfies a given condition for chunking (e.g. $fp \bmod 2^{12} = 0$), CDC splits the input stream at this position. For each byte of sliding, CDC should calculate the fingerprint of the sliding window and test the chunking condition, and thus causes extremely high performance overhead. To reduce the cost, the Rabin fingerprinting scheme [9] can calculate new fingerprints based on the previous one. AE [21] explores the asymmetrical local range to identify cut-points, and thus improves the throughput of chunking. FastCDC [22] improves the performance of chunking by simplifying the condition testing and carefully choosing cutting points.

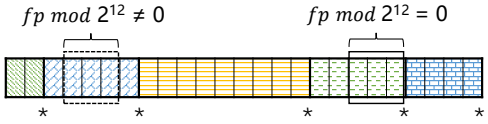


Fig. 2: Content-defined chunking with a sliding window of 3 bytes. *The data in this example is split into 5 chunks, and the cutting points are marked with stars.*

After chunking, fingerprints are indexed to facilitate the identification of duplicate data. Most previous proposals maintain a large portion of fingerprints in disk, and only keep the most recently used fingerprints in memory. To alleviate the cost of frequent accesses to disk, Aronovich et al. [23] exploit the similarity of backup streams and large chunks (16 MB) to reduce the size of fingerprints. DDFS [24] and SiLo [11] achieve a similar goal by exploring the similarity and spatial locality of data streams. ICID is completely different from these studies because it does not use hash functions to identify redundant data.

Data deduplication systems can be implemented in different layers of the I/O stack. CAFTL [18] and CA-SSD [25] achieve deduplication in *flash translation layer* (FTL) of SSDs to reduce the wear-out and prolong their lifespan. Dmdedup [26] is implemented in the block layer with fixed-size chunking. NV-Dedup [20] achieves deduplication for an NVM-based filesystem. FPC [27] integrates a deduplication scheme into EXT4/F2FS filesystems to reduce the write traffic. ICID offers a deduplication library that can be flexibly integrated with different applications, such as KV stores and text/multimedia editors.

There have been only a very few studies on data deduplication by exploiting the causality among I/O operations. NLE-DFFS [28] is a deduplication system designed for flash-based filesystems. It mainly targets to non-linear editing (such as video editing) in an embedded system. However, because it maintains all metadata of memory operations in a list and lacks an effective method to manage/reclaim the metadata, the effectiveness of NLE-DFFS is not clear in a large-scale storage system. Provenance-aware storage systems [29] maintain the lineage of different files by tracking command line and system calls. They are mainly used for debugging, auditing, intrusion detection. Unfortunately, they lack sufficient information about data redundancy among files. Inspired by those proposals, we take the first step to explore I/O causality based data deduplication for NVMs, and develop a deduplication library for applications.

3 Motivations

We observe that a large amount of duplicated data are caused by file processing applications, such as office software, video editors, and KV store systems. These applications may frequently update (i.e., insert/merge/truncate) a portion of data in existing files. Because of the semantic gap between applications and filesystems, the filesystem treats the updated file as new data and write all content to storage directly. Since the filesystem is oblivious to the partial redundancy between the source file and the target file, it often causes unnecessary write traffic and significant write amplification. Traditional chunk-based deduplication

systems all overlook the rich I/O semantics of applications. They usually suffer significant performance and storage overhead due to the fingerprint computation and indexing.

In this paper, ICID tries to eliminate the semantic gap between application and filesytems by tracking the data lineage between files, and achieves fine-grained data deduplication by fully exploiting the promising features of NVMs, i.e., byte-addressability and high random read bandwidth relative to SSD/HDD. For example, if file A is generated by combining file B and file C through memory-copy operations, we can track file A's lineage by recording these memory-copy operations in a special data structure. When file A is written back to an NVM-based file system, we can exploit the data lineage to eliminate data redundancy among file A, B, C, and only write the metadata of file A rather than its content to the NVM. We call this approach as I/O causality based in-line deduplication. It can significantly reduce the data traffic written to NVMs, and improves data deduplication ratio. However, our approach still faces several challenges.

First, *how to organize the records of memory-copy operations with low cost of lookup?* As all I/O causality relationships are recorded in a tree-like data structure called *rec-tree*, ICID needs to search the rec-tree to update or retrieve the I/O causality for each copy/write operation. Since these operations are frequently performed in most programs, the search procedure should be as fast as possible. Thus, we should carefully organize the memory-copy records in a data structure and manage them efficiently.

Second, *how to identify and abandon outdated records in the rec-tree efficiently?* As the program continues to run, more and more records are stored in the rec-tree. As time goes by, most records become useless because the program's data corresponding to the outdated record may be updated or overwritten. These outdated records should be abandoned in time because they waste memory space. Moreover, massive records also increase the latency of the B-tree lookup. However, it is challenging to identify outdated records accurately and efficiently.

Third, *how to identify the duplicate portion efficiently when the data is written to NVM?* When the file data is copied to memory, the data may be updated or overwritten. Because it is costly to track all memory updates, ICID compares the written data with the original file to identify the redundancy. Since byte-by-byte data comparison is usually costly, it is essential to design an efficient approach for data comparison.

4 Design and Implementation

In this section, we present the design details of ICID. To achieve high deduplication ratio while minimizing the performance overhead, we propose three key techniques, i.e., organizing memory-copy records in a B-tree structure at the page granularity, garbage collection for outdated memory-copy records, and data comparison using SIMD hardware.

4.1 Overview

To bridge the semantic gap between applications and filesystems, we develop a user-level I/O library to support in-line data deduplication during file operations. ICID defines a set of filesystem interfaces similar to POSIX I/O interfaces, such as **`ICID_open()`**, **`ICID_read()`**, and **`ICID_write()`**. The difference between ICID and POSIX for programmers is that the name of ICID APIs starts with a prefix "**`ICID_`**". The key APIs in ICID are listed in Table 2.

TABLE 2: Key APIs in the ICID library

Operations	APIs
file open/close	int **`ICID_open`** (const char *path, int oflag, ...) int **`ICID_close`** (int fd)
memory copy	void ***`ICID_memcpy`** (void * dest, void * src, size_t len)
memory compare	void ***`ICID_memcmp`** (void * s1, void * s2, size_t len)
memory map/unmap	void * **`ICID_mmap`** (void *addr, size_t len, int prot, int flags, int fd, off_t offset) int **`ICID_munmap`** (void *addr, size_t len)
file read/write	size_t **`ICID_read`** (int fd, void *buf, size_t len) size_t **`ICID_write`** (int fd, void *buf, size_t len)

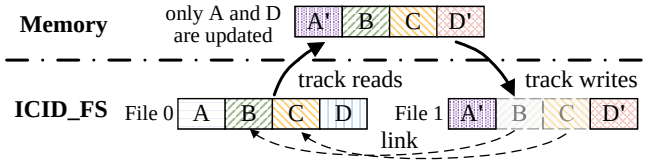


Fig. 3: An example of data deduplication in ICID

For the file open operation (**`ICID_open()`**), ICID not only creates a file descriptor, but also maps (i.e., **`ICID_mmap()`**) the file to main memory. In the later, the file read (e.g., **`ICID_read()`**) is actually replaced by a memory-copy operation. We use *copy* in the following to refer to both traditional *copy* and *read* operations. A B-tree (called rec-tree) is created to maintain the metadata of memory-copy operations for a single process. Each record in the rec-tree corresponds a memory-copy operation. ICID relies on these memory-copy records to get hints about the potential duplicate data. Before the updated file data is written back (**`ICID_write()`**) from a memory buffer to storage, ICID checks whether the data is copied from existing files by querying correlated records in the rec-tree. If the original data is copied from an existing file, ICID exploits SIMD hardware to further check which portion of data is completely the same as the original file's. Finally, ICID updates the metadata of the redundant data, and write back data slices without any redundancy.

Figure 3 illustrates an example of data deduplication in ICID when a file is updated and saved as another file. When File 0 is loaded into memory, ICID tracks I/O operations to detect data redundancy. When the updated File 0 is written back, ICID only writes the updated data block A' and D' to storage, and the unmodified data blocks (B and C) are linked to their original versions in File 0. In this way, only modified data blocks in memory are written to storage, thereby saving storage space.

Figure 4 illustrates the detailed workflow of data dedu-

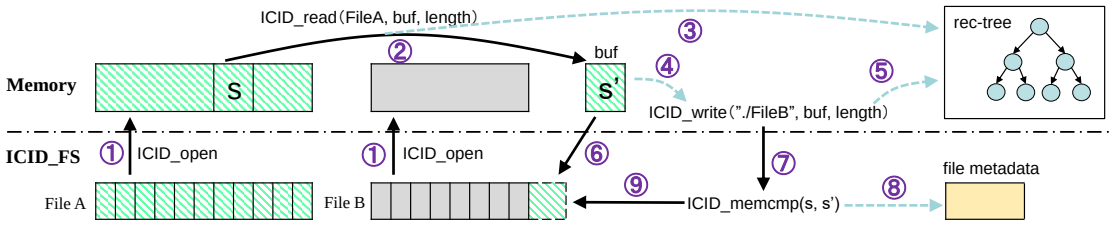


Fig. 4: A illustration of file appending in ICID. *This example shows how ICID detects data redundancy when a program appends a portion of data from file A to file B. The solid and dotted arrows represent the flow of data and metadata, respectively.*

plication in ICID. We append a portion of file A to file B. Both file A and file B are stored in our user-level filesystem called ICID_FS. Because the appended data is copied from file A, it can be deduplicated inline. At first, files A and B are opened and transparently mapped to main memory (①). When the program reads (copies) a portion of data (S) from file A, ICID first copies the data to the destination buffer (②), and then records the memory-copy operation in the rec-tree of the program (③). When the data in the memory buffer (S') should be written back (④), the program calls **`ICID_write`**. ICID searches the rec-tree to check whether the data S' is originally copied from existing files (⑤). If the data in the buffer is not copied from other files, the data S' is directly appended to file B in ICID_FS (⑥). Otherwise, **`ICID_memcmp`** (⑦) is performed to inspect the duplicate portion because the data (S') may be updated by the program. If the data S' is exactly the same as S, the metadata of file B is updated (⑧), and thus avoid writing the content to the storage. Otherwise, the data content is written back, similar to the ordinary file write operation (⑨).

Figure 5 shows the pseudo code of the example illustrated in Figure 4. In lines 4-5, ICID opens the file and maps them to main memory silently. In line 7, **`ICID_read`** actually uses a memory-copy operation instead of a read operation in the background to copy the file content to the buffer, and then inserts the memory-copy record into the rec-tree. In line 8, **`ICID_write`** identifies the data redundancy and writes the data (or metadata) back.

We note that these ICID APIs are only used for I/O operations that may cause data redundancy, such as file copying/merging/truncation etc., because these operations have to rewrite existing file data into storage. For other file operations, no matter the file is deduplicated or not, the traditional file APIs are used, without accessing the rec-tree. Moreover, ICID APIs are applicable for both in-memory and on-disk file systems no matter the NVM is used as main memory or *directly access* (DAX) supported storage devices. Although ICID can be also used by traditional SSD/HDD devices, the fine-grained deduplication may cause sever read/write amplification due to block-based I/O operations. Thus, the ICID library is particularly beneficial to NVM-based storage systems.

ICID manages one rec-tree for each program, and achieves data deduplication when a program is editing files. However, when the deduplicated data is written back to storage, its metadata is still managed by the file system, and is visible to all programs globally. Thus, the scope of dedu-

```
1.  int appendFiles(char *fileA, char *fileB){
2.      int fd_A, fd_B;
3.      char *buf;
4.      fd_A = ICID_open(fileA, O_RDONLY);
5.      fd_B = ICID_open(fileB, O_RDWR, O_APPEND);
6.      buf = malloc(fd_A.size);
7.      ICID_read(fd_A, buf, fd_A.size);
8.      return ICID_write(fd_B, buf, fd_A.size);
9.  }
```

Fig. 5: File appending using ICID interfaces.

plication is per-program execution, but the deduplication effects are propagated to the file system globally.

The rec-tree is only stored in main memory. When a program is running, the rec-tree is used to record all memory-copy operations for in-line deduplication. When the program ends, the rec-tree is deleted and the consumed memory is reclaimed. We do not need to persist the rec-tree because it becomes useless when the data is written to the storage. Upon a power failure, it is unnecessary to recover the old rec-tree because it becomes valueless, and a new rec-tree will be rebuilt when the program restarts.

4.2 Data Structure of Memory-Copy Records

To look up memory-copy records efficiently and facilitate the garbage collection of outdated records (§ 4.3), ICID uses a resizable array to store memory-copy records whose destination addresses are related to the same page in ascending order, and manages each resizable array in a node of a B-tree. For each program, all memory-copy records are managed in a B-tree-like data structure (called rec-tree). The address range of each tree node corresponds to one page (4 KB), and memory-copy operations involved in a page are recorded in the corresponding tree node. Since multiple threads may simultaneously read/modify the rec-tree, we use a readers-writer lock to guarantee exclusive access to a data node in the B-tree. For each memory-copy operation, such as **`ICID_memcpy()`** and **`ICID_read()`**, ICID generates a new memory-copy record. A record contains both the source and destination addresses to identify a unique memory-copy operation. If the data is read/copied from existing files, the record is inserted into the rec-tree. As shown in Figure 6, the leftmost rectangle shows the structure of a memory-copy record. The file descriptor and the file offset (*offsetInFile*) together specify the source address of the memory-copy. The page number and the page offset

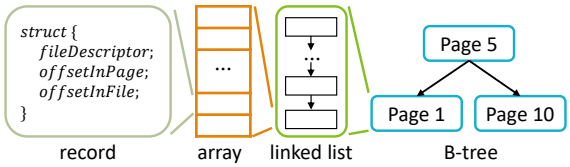


Fig. 6: The data structure of rec-tree

(*offsetInPage*) together determine the destination address. The page number is stored in the node of the B-tree because it is shared by multiple records.

The rec-tree is frequently accessed by ICID APIs due to two reasons. First, for each memory-copy operation, ICID should check whether the memory-copy comes from a mapped memory area. If not, ICID searches the rec-tree to check whether the data is copied from existing files. Second, for each write operation, ICID also has to search the rec-tree to check whether the written data is redundant. However, frequent memory-copy operations imply that massive memory-copy records are generated. If we simply manage these records in a pure B-tree structure, the record lookup is often costly.

To mitigate the cost of record lookup, we advocate a coarse-grained B-tree in which a data node is composed of a resizable array (or array list), as shown in Figure 6. All records whose destination addresses target to the same page are clustered in a single node of the B-tree, i.e., these records in a single node are indexed by the same page number. Since the number of records in a node is variable, we use two lightweight data structures (i.e., linked lists and arrays) to store these records.

We note that it is inefficient to use only a linked list or a array solely because they both have disadvantages. Linked list is efficient for insert/delete operations, but shows poor performance of data traversal due to pointer chasing, i.e., each data node causes extra memory accesses to pointers when traversing the linked list. In contrast, an array can be quickly accessed by simply refereeing to its index number, but its immutable size lacks flexibility. The memory space must be allocated with the array's size during the initialization. In our case, assume three tree nodes contain 2, 3, and 20 records, respectively, and each record is 16 bytes, 320 (i.e., 20×16) bytes are required to store each record using arrays solely. Thus, total 560 (320×3-(2+3+20)×16) bytes of memory space are wasted, and only 41.7% of the total array space is used.

To fully exploit the advantages of both linked lists and arrays, we integrate linked lists with arrays to implement a resizable array. As shown in Figure 6, we organize all records in a three-level data structure including B-tree, linked lists, and arrays. Inside the tree node, each element of the linked list contains an array, and each record in the array represents a memory-copy operation. The node of B-tree keeps both the tail and head of the linked list. For each insertion, ICID directly writes a record into an array if it is not fully used. Otherwise, ICID first allocates a new array for the linked list, and then insert the record into the array.

Our three-level data structure is cost-efficient for both data retrieval and memory consumption. First, organizing memory-copy records in the page granularity reduces the number of tree nodes, and thus reduces the cost of data searching. Second, when a memory-copy record is traversed by ICID, our data structure offers higher performance for data traversal in arrays. The combination of arrays and linked lists also reduces memory consumption. In the same situation as the previous example (three tree nodes with 2, 3, 20 records respectively), if we set the array size to 6, our approach only needs 5 arrays ($\lceil 2/6 \rceil + \lceil 3/6 \rceil + \lceil 20/6 \rceil$), causing 176 bytes memory waste (16×6×5-(2+3+20)×16 bytes), less than one third of the array.

Since ICID may frequently insert/delete memory-copy records in the rec-tree, the size of array list may be changed dynamically. We use a reserved memory pool to reduce the cost of memory allocation for the array list.

4.3 Garbage Collection of Memory-Copy Records

It is essential to delete outdated memory-copy records in time from the rec-tree. The reasons are as follows: 1) Outdated records are valueless. These records increase the amount of nodes in the memory, and thus increase the latency of traversal. 2) Outdated records unnecessarily consume memory capacity. 3) Outdated records often provide fallacious hints about the duplicate data. They lead to unnecessary data comparison (§ 4.4) which increases the cost of data deduplication.

There are mainly two scenarios that memory-copy records become outdated. First, if a new record's destination address overlaps with an old one's, the old one is deemed as an outdated record. Second, after a memory-copy operation, some operations such as variable assignment or **`memset()`** may overwrite the copied data. In this case, the previous memory-copy record becomes an outdated record. However, it is difficult to identify these outdated records. Since memory-copy operations are often conducted in a continuous memory space, it needs range queries to identify these outdated records. A simple approach is to traverse existing records before inserting a new record, and find out records whose destination address overlaps with the new one. In this way, only the first kind of outdated records can be detected, and thus a traverse of records is required for each memory copy, unnecessarily increasing the performance overhead.

To figure out the pattern of memory-copy operations, we examine three applications, i.e., Patch [30], FFmpeg [31] and LevelDB [32]. We find that *the destination addresses of adjacent memory-copy operations mainly follow an ascending order, i.e., most destination addresses are increasing*. The reason is that the virtual memory is often allocated to process from the low address to the high address. This observation also fits for the programming habit for most programmers. Based on this observation, we have Inference 1:

> **Inference 1:** *In a single continuous memory region, if the destination address of a newly-inserted memory-copy record is lower than others', the record with a higher destination address is expected to become outdated in the near future.*

Since deleting those records that would become outdated soon has a little impact on the deduplication ratio, we advocate Optimization 1 to delete outdated records.

Optimization 1: the memory-copy record whose destination address is higher than the newly-inserted one can be deleted. Since each node only contains memory-copy

records within a 4 KB page, ICID does not delete records whose destination addresses are beyond the current page address. Upon each insertion, ICID first finds the node in the rec-tree according to the page number. Second, ICID traverses the array/linked list structure inside the node, and deletes all records whose destination address is equal to or higher than the newly-inserted one's. Third, ICID relocates the following records to fill the hole caused by the deletion. At last, ICID adds the record to the tail of the array/linked list structure. In this way, ICID is able to delete most outdated records in time. However, traversing and relocating records inside tree nodes upon each insertion often cause non-trivial performance overhead. Fortunately, these traversal and relocation operations are unnecessary because the Optimization 1 can guarantee all records in the array list are ordered, as described in Inference 2.

> **Inference 2:** *All memory-copy records inside a single node of the rec-tree are arranged in ascending order of the destination address.*

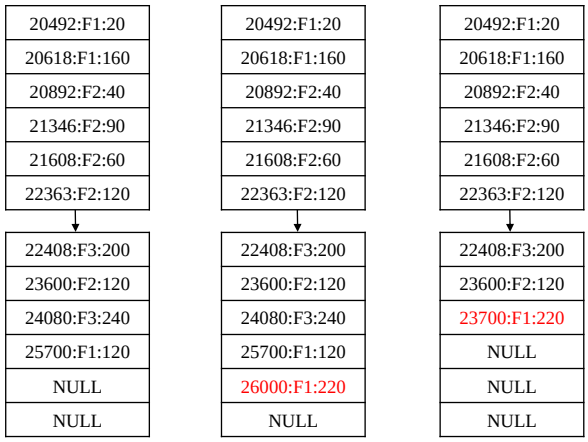
Proof. For each newly-inserted memory-copy record e_i and its destination address $D(e_i)$, Optimization 1 guarantees that the record e_i has the highest destination address in this array list, i.e., $D(e_j) < D(e_i)(1 \leqslant j < i)$. Since all records are inserted in this way, for each record $e_k(k > 1)$, it satisfies $D(e_{k-1}) < D(e_k)$. □

Based on Inference 2, we have **Optimization 2: all memory-copy records in each resizable array can be sorted in ascending order if we directly delete outdated memory-copy records upon each insertion, without causing any data movement.** As shown in Figure 7, for each insertion of the memory-copy record, ICID first finds the node in the rec-tree according to the page number. Then, ICID compares the new record's destination address with the last record in the array list (i.e., 25700 in Figure 7(b)). If the new record's address is higher, ICID appends the record to the tail of the array list (Figure 7(b)); Otherwise, ICID takes the following steps:

1) traverses the array list to find the insert position, i.e., the first record whose destination address is higher than the newly-inserted one's;
2) deletes all records after the insert position;
3) write the new record at the insert position (Figure 7(c)).

However, traversing the array list to find out the insert position is still time-consuming. ICID can accelerate the traversal because all records in the array list are arranged in an ascending order (Inference 2). ICID only compares the first element of adjacent arrays to find out an array that contains the insert position quickly. Then, it traverses the array to locate the insert position. For example, in Figure 7(c), before inserting the record of 23700, ICID first compares 20492 and 22408 with 23700, and then traverses the array from the record 22408 till it finds the record 24080.

We note that ICID may delete useful memory-copy records in a few cases. Thus, ICID may lose a few opportunities to find duplicate data. However, the wrongly-deleted memory-copy records do not affect the correctness of applications because the rec-tree is only used to find duplicate data. Our evaluation in § 5.2 shows that ICID achieves rather high deduplication ratio, implying that Inference 1 is well fit for most real-world cases. We also note that when a mapped file is unmapped, ICID deletes all records corresponding to the file from the rec-tree. In this way, ICID ensures that no outdated memory-copy record can survive in the process' memory space forever.



(a) Sorted structure (b) Insert the record of 26000 (c) Insert the record of 23700

Fig. 7: Data management in the resizable array. *Memory-copy records are represented in the following format:* ***[destination address : file : file offset]****. Sub-figure (a) shows the ordered structure after using Optimization 1, Sub-figure(b) and (c) shows the data structure after inserting memory-copy records with the destination addresses 26000 and 23700, respectively. Because 26000 is higher than all records, it is appended to the tail. Before inserting the record of 23700, ICID deletes all records whose addresses are higher than 23700 (24080, 25700, and 26000).*

TABLE 3: Time consumption of writing data to NVM and data comparison (byte-by-byte and AVX2)

Data size (Bytes)	Time consumption (ns)		
	write to NVM	byte-by-byte	AVX2
128	1782	201	41
256	1785	358	38
1024	4986	3679	77
4096	8667	15484	249
8192	14260	31294	469

ICID offers several advantages as follows. First, the data volume of records is reduced significantly, and thus reduces the cost of record traversing during data deduplication. Second, the record traversal is only needed when the destination address of newly-inserted record is lower than the one at the tail. Third, for each traversal of the array list, ICID only needs to access a small portion of records, and thus further reduces the traversal cost. Fourth, no data relocation is required because we always delete outdated memory-copy records from the tail of linked list.

4.4 Data Comparison

When a file is copied to a memory buffer, the content may be changed by applications. Since the I/O causality derived from memory-copy operations only offer hints to identify potential redundancy between the original file and

the data to be written, ICID should compare the data with the original file to figure out the real duplicate parts before the data in the buffer is written back to storage.

Table 3 shows the time consumption of writing data back to NVM and byte-by-byte comparison. When the data size is bigger than 4 KB, the time consumed by the byte-by-byte comparison is even longer than the latency of writing data back to the NVM, indicating the inefficiency of directly comparing. ICID explores SIMD hardware to accelerate the data comparison. With AVX2 hardware extensions, ICID can compare 256 bits of data in a single instruction. Taking 4 KB data as an example, AVX2 is 62 times faster than the byte-by-byte comparison. ICID compares the data in the buffer with the original file from the front to the end. If it finds a different byte between the buffer and the original file, the comparison stops and ICID calculates the length of duplicate data. If all content in the buffer is verified as the same as the original file, the whole buffer is deemed as a duplicate data.

4.5 File Organization

We implement ICID in an NVM-based storage system for deduplication, called NVMDedup. In the following, we mainly introduce how deduplicated files are managed. Generally, a file in NVMDedup is often composed of a number of extents. Each extent identifies a continuous area in the storage through an offset and an length. The physical addresses represented by extents in different files may be overlapped. Since a file may contain multiple small extents, the number of extents in a file may be very large. To accelerate the file read operation, NVMDedup organizes all extents of a file in a unique B-tree. When a file is updated, NVMDedup exploits Copy-On-Write (COW) mechanism to guarantee data integrity because a data slice may be shared by multiple files. The in-place updating mechanism may face a risk of data corruption in the presence of a power failure or a system crash. As shown in Figure 8, assuming the extent A is shared by multiple files. When the extent A is updated by File N, it is cloned and updated as a new extent A'. However, A' is only referenced by File N, while other files still refer to the old extent A. As a result, each program can only see the new version of data updated by itself, while the original version of data remains unchanged. ICID_FS uses a journal to protect the metadata in the presence of system crashes. The journaling mechanism has been widely used in a variety of file systems [33], [34]. Before updating the metadata, ICID_FS stores the new metadata in a log region. After the metadata has been updated successfully, the log is committed and marked for deletion. When a system crashes and restarts again, ICID_FS checks the log region and redo the uncommitted log. This journaling mechanism can guarantee metadata consistency upon system crashes. With COW and journaling mechanisms, ICID guarantees the consistency of both data and metadata.

4.6 Garbage Collection for ICID_FS

In this section, we briefly introduce the garbage collection (GC) of file systems. We note that it is different from the garbage collection of rec-tree. When a file is deleted in ICID_FS, only the metadata of this file is deleted because

(a) An extent shared by multiple files (b) Extents after an update

Fig. 8: An illustration of the copy-on-write mechanism

the corresponding extents may be referenced by other files. Thus, a few extents may be not referenced by any file, and becomes garbage. ICID_FS should reclaim the storage space consumed by those unreferenced extents periodically. During the GC process, the GC thread scans all metadata and calculates the reference count of each extent. All extents with zero reference are deleted. The GC thread also performs data compaction to eliminate fragmented storage space. During compaction, different extents are compacted in continuous physical addresses, and thus multiple smaller free regions are consolidated into larger ones. This process continues till the consolidated free space becomes larger than 4 KB. When an extent is being moved by the GC thread, we use locks to block any accesses to this extent, and thus can guarantee data integrity and consistency during GC. To minimize performance impact on user applications, NVMDedup triggers the GC process when the CPU load is low. However, when the available storage space is lower than a given threshold, the GC should be performed soon.

5 Evaluation

In this section, we conduct experiments to demonstrate the efficiency of ICID. Compared with the state-of-the-art chunk-based deduplication systems, ICID shows better performance on data deduplication for in-memory file operations: ICID achieves higher deduplication ratios, and also improves the efficiency of data deduplication.

5.1 Experimental Setup

System Setup. We run experiments on a real-system equipped with two Intel Xeon Gold 6230 CPUs running at 2.1 GHz and 128 GB DDR4 memory. If not specified otherwise, all experiments use 128 GB Intel Optane DCPMM DIMMs in the *fsdax* mode. The operating system is Ubuntu 19.01 with a kernel version 5.1.1. ICID runs on ICID_FS, while chunk-based deduplication approaches use EXT4 filesystem mounted with *rw, realtime* options.

Benchmarks. We evaluate ICID with 4 real-world benchmarks: LevelDB [32], GNU Patch [30], FFmpeg [31], and Download. They are representative applications of key-value store, version control, video editing, and network transmission. LevelDB is a popular key-value (KV) store that stores KV pairs in *Sorted Strings Tables* (SSTables). LevelDB uses LSM-tree to organize SSTable files in multiple levels. As the size of the KV store increases, the data volume in a single level would exceeds its storage limitation. At this time, LevelDB picks multiple files and compacts them into a new and larger file, and then stores it in the next level. Since the newly-generated file comes from existing files, there are quite a lot data redundancy among these files. GNU Patch applies patch files to one or more original files. FFmpeg is a multimedia library that underpins the media services of YouTube and iTunes. LevelDB, GNU Patch, FFmpeg all

TABLE 4: The deduplication ratio of different schemes

Schemes	Deduplication Ratio			
	LevelDB	Patch	FFmpeg	Download
FSC [19]	4.58%	0.02%	0.09%	3.3%
AE [21]	4.50%	0.02%	0.06%	13.7%
RABIN [36]	4.40%	0.00%	0.07%	13.9%
FastCDC [22]	4.60%	0.02%	0.10%	14.3%
ICID	81.36%	98.43%	53.24%	0%

clip or merge existing files into new files, and thus usually cause high data redundancy. To apply ICID APIs to these benchmarks, we modify about 60, 30, 10 lines of code (LOC) in LevelDB, FFmpeg, and Patch, respectively.

For LevelDB, we use its native benchmark–db_bench and disable the content compression. By default, this benchmark writes one million randomly-generated KV pairs with 16-byte keys and 100-byte values into the KV store. In our experiments, we mainly use LevelDB as our benchmark because it can easily change the dataset by configuring the number of KV pairs and value size. For FFmpeg, we clip the first half of a video. The container format of the file is avi, with video encoded in MPEG-4 and audio encoded in AC-3. Both the container format and video/audio encoding are widely used in industrial production systems and the daily life. For GNU Patch, we download different versions of Linux kernel sources and their patches to patch them. In our evaluation, we only find data duplication between the patched file and the original file. We also construct a benchmark called "download" to evaluate a scenario where the input data comes from Internet. Each scheme downloads 3 GB of the same dataset comprising papers and slides.

Systems for Comparison. To compare ICID with chunk-based deduplication schemes, we design a general framework to track data to be written and split it into chunks which are deduplicated using a similar approach like NV-Dedup [20]. We use XXHash [35] to calculate fingerprints of these chunks for high throughput. We compare ICID with four typical chunk-based schemes: FSC [19], Rabin [36], AE [21], and FastCDC [22]. We integrate these approaches into the I/O stack by intercepting *write* system calls, and then measure the deduplication ratio and the execution time of different workloads. The chunk size is 4 KB by default.

To demonstrate the advantages of our key designs for I/O-causality based deduplication, we also implement another version of ICID called ICID_DO for comparison. Unlike the full implementation of ICID, ICID_DO deletes all records whose destination addresses overlap with the newly-inserted one upon each insertion. This simple garbage collection scheme is memory efficient, but may incorrectly delete useful memory records.

5.2 Deduplication Ratio

We evaluate the dedudplication ratio of different dedudplication schemes using four real-world applications. The results are shown in Table 4. For the first three benchmarks, ICID achieves the highest deduplication ratio compared with chunk-based deduplication schemes. The deduplication ratio of chunk-based schemes is lower than 5%. The root causes are two-folds. First, the small size of written

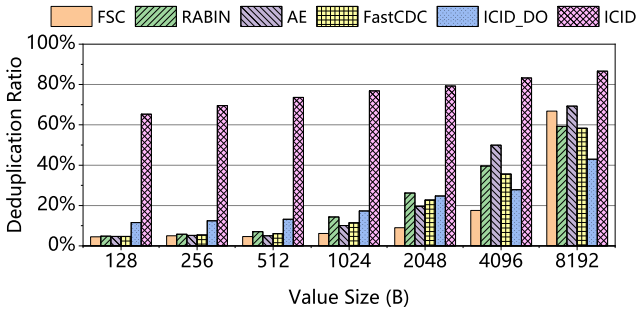


Fig. 9: The data deduplication ratio in LevelDB

data in these applications limits the effectiveness of chunk-based schemes. Patch and FFmpeg tend to write a small amount of data (e.g., 1024 bytes) in a single I/O operation. Then, those small inputs are chunked by chunk-based schemes. Since the expected chunk size is larger than the input, these chunk-based schemes treat the whole input as a single chunk. In other words, chunk-based schemes are not effective for those small inputs. Second, since chunk-based schemes have no application semantics to hint the data duplication, they may mix the duplicate data and the unique data into the same chunk, and thus achieve low deduplication ratios. In contrast, ICID records the I/O-causality of in-memory file operations, and uses them as hints for data deduplication in a more fine-grained manner. Thus, ICID achieves the highest deduplication ratio. For download benchmark, ICID shows a deduplication ratio of zero because there is no I/O causality among these downloaded files, and thus cannot identify the redundant data. In contrast, other chunk-based deduplication schemes show reasonable deduplication ratios. These benchmarks demonstrate that there is not a one-size-fits-all approach for different scenarios. ICID is particularly effective for processing local files, while chunk-based deduplication schemes are more applicable to other scenarios.

To evaluate the sensitivity of the deduplication ratio to the size of duplicate data, we run LevelDB with various value sizes. As shown in Figure 9, ICID achieves the highest deduplication ratio for all cases. Since the redundancy of LevelDB is mainly caused by rewriting key-value pairs during compaction, larger value sizes cause more redundancy, and thus result in higher duplication ratios for all schemes. ICID can achieve rather high duplication ratios even when the value size is smaller than the chunk size (4 KB). Benefiting from I/O-causality, ICID_DO also shows higher deduplication ratios than chunk-based schemes when the value is smaller than 2 KB. ICID can improve the deduplication ratio by up to $16\times$ compared with other schemes. Because ICID can identify and delete outdated memory-copy records in the rec-tree more correctly, it can further improve the deduplication ratio.

In most cases, FSC shows the lowest deduplication ratio as it does not consider the content of the input during chunking. However, for the value size of 8,192 bytes, FSC shows a higher deduplication ratio than CDC-based schemes except AE. The reason is that the written data contains some metadata which are unlikely to be duplicated, for example, CRC, filter block, and footer [32]. These metadata

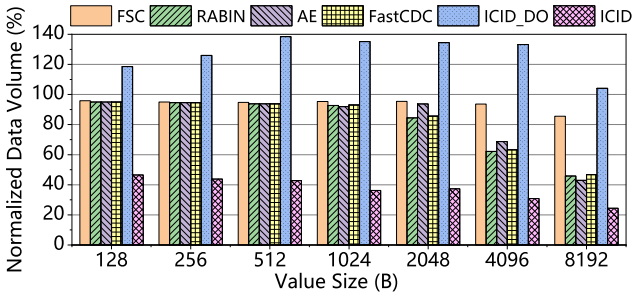


Fig. 10: Write traffic volume of different approaches. Data is normalized to LevelDB with compression enabled.

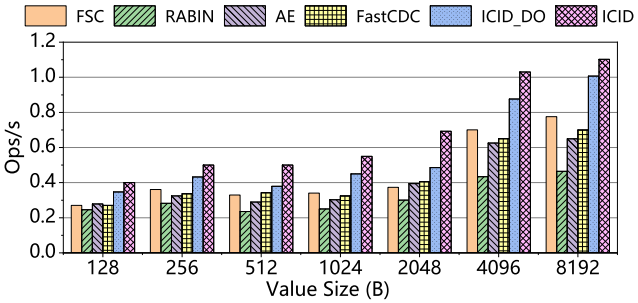


Fig. 11: The impact of data duplication on the IOPS of LevelDB, all results are normalized to the IOPS of standalone execution without deduplication.

are usually placed in the tail of the buffer. Because the value size can be divided by the chunk size, FSC is likely to place the metadata in a single chunk. Unlike fixed size chunking, CDC-based schemes perform chunking based on the content of data, and thus have a higher possibility to mix KV pairs and the metadata (i.e., the duplicate and unique data) into the same chunk, causing a lower deduplication ratio.

5.3 Write Traffic Reduction

ICID detects data redundancy by tracking memory-copy operations, and thus cannot handle compressed data. As a result, ICID has to disable content compression in LevelDB. In the following, we verify whether the finer-grained deduplication in ICID can reduce more write traffic than the content compression in LevelDB. Figure 9 shows the write traffic of different approaches, all normalized to the vanilla LevelDB with content compression enabled. For chunk-based approaches, we enable the content compression in LevelDB. ICID achieves about 60% write traffic reduction on average because of its high deduplication ratios. Chunk-based approaches achieve much less write traffic reduction because of their low deduplication ratios. ICI_DO results in larger write traffic than the vanilla LevelDB because ICID_DO leads to less write traffic reduction than content compression in LevelDB. However, ICID_DO is just used for comparison, it is impractical for real-world scenarios. Overall, even ICID disables the content compression, it can still achieve significant write traffic reduction compared with chunk-based approaches.

We also analyze the impact of write traffic reduction on the life time of NVM devices. Generally, the lifetime of NVMs is proportional to its capacity and is inversely proportional to the data volume written on it. Assume C is the capacity of the NVM device, D is the volume of the original write traffic, and R is the deduplication ratio. We use the following equation $\frac{C}{D\times(1-R)}$ to estimate the lifetime of NVM devices. According to the write traffic volume shown in Figure 10, ICID can prolong the lifetime of NVM devices by 2.1–4.1 times.

5.4 Throughput

To better understand the impact of ICID on the performance of different workloads, we run LevelDB with different value sizes, and compare *I/O operations per second* (IOPS) of LevelDB with the standalone execution without data deduplication. As shown in Figure 11, all results are normalized to the IOPS of LevelDB without deduplication. Our experiments show that the average IOPS of vanilla LevelDB is 230 KOps/s, with an average latency of 4.3 us. When the value size is as small as 128 bytes, ICID lowers the IOPS of LevelDB by 60%, while other chunk-based data duplication schemes degrade the application performance even by 76%. The performance overhead of ICID is mainly attributed to the lookup cost of the rec-tree. With the growth of value size, the performance of ICID gradually approximates to that of the vanilla LevelDB. When the value size is as large as 8 KB, ICID can even improve the throughput by about 10% because the benefit of write traffic reduction eventually exceeds the cost of data deduplication. ICID spends similar time to detect the potential redundancy in the rec-tree, regardless of the size of I/O operations. However, for larger I/O sizes, ICID can detect more redundancy at one time. Thus, ICID achieves more data traffic reduction for large KV pairs while the cost of redundancy detection is similar to that of small KV pairs. We find that the throughput of chunk-based schemes is also improved when the value size increases. The reason is that larger value sizes lead to higher deduplication ratios and less write traffic.

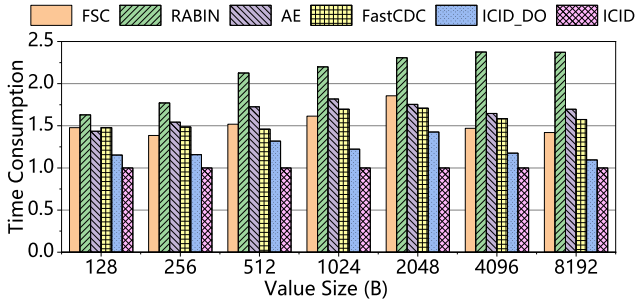


Fig. 12: The deduplication time of different schemes varies with the value size in LevelDB, all normalized to ICID.

5.5 Efficiency of Data Deduplication

In this section, we evaluate the efficiency of different data deduplication schemes by measuring the execution time of data deduplication. To make a fair comparison, we write the original data on the file system no matter whether we find out the duplicate data slices, so that all deduplication systems spend almost the same time on file read and written. We run LevelDB with various value sizes and measure

the execution time of workloads. As shown in Figure 12, ICID can reduce the time cost of data deduplication by 22% on average compared with ICID_DO. This implies that the rec-tree is efficient for reducing the runtime overhead of deduplication.

Compared with other deduplication schemes, ICID reduces the execution time of workloads by 47% on average. ICID is the most efficient deduplication scheme for all workloads. The reason is that ICID can significantly decrease the search space of data deduplication via tracking I/O-causality among in-memory file operations. ICID records memory-copy operations in a rec-tree, and then exploits these records to find out the hint about data redundancy. Thus, ICID can significantly reduce the the cost of data deduplication compared with traditional chunk-based schemes. Moreover, ICID exploits an efficient garbage collection scheme to delete outdated memory-copy records, and further shrinks the search space of data deduplication. FSC shows much shorter execution time than other chunk-based schemes because it chunks the input file in a fixed size, and thus avoids the time-consuming cut-point calculation. However, FSC usually shows lower deduplication ratio, introduces more data chunks, and spend more time on searching the indexes.

To better understand the performance overhead of ICID and chunk-based deduplication approaches, we run LevelDB with the value size of 4 KB and measure the execution time of different steps in these two approaches. As shown in Figure 13, chunk-based approaches spend about 66% more time than ICID to deduplicate the same volume of data. ICID spends about 82.4% of total execution time on maintaining memory-copy records. Since ICID leads to an insertion or a deletion in the rec-tree for each memory-copy which is often a frequent operation in many file-processing applications, it is not surprising that the majority of time is spent in maintaining these records. ICID only spends approximately 15.3% of total execution time to search the potential redundancy in the rec-tree for write operations which are much less frequent than memory-copy operations. The data comparison accounts for only 2.3% of total execution time because ICID can significantly accelerate this operation through AVX instructions. For chunk-based deduplication approaches, the chunking and hashing operations spend most of total execution time. Chunking is particularly time-consuming because it has to calculate a rolling hash for each byte of input. Hashing also accounts for a large amount of time due to its high computation complexity. Overall, chunk-based approaches are much more costly than ICID.

5.6 Sensitivity to the Chunk Size

To evaluate the impact of the chunk size on the deduplication ratio and the deduplication time, we run LevelDB with one million unique key-value pairs using different chunk sizes. The key and value sizes are set to 16 and 1024 bytes, respectively. As shown in Figure 14, when the chunk size is smaller, although the deduplication ratio is higher, chunk-based schemes suffer unacceptable performance overhead in terms of long deduplication time due to hashing and indexing. Thus, it is often impractical to use very small chunk sizes for in-line deduplication. Most deduplication systems

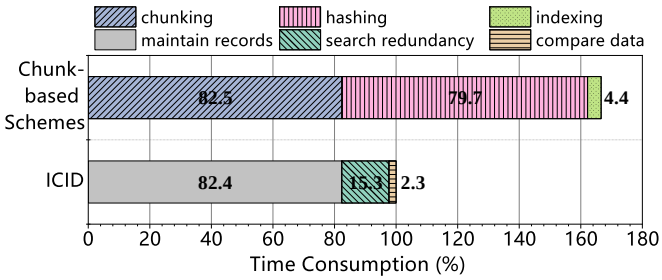
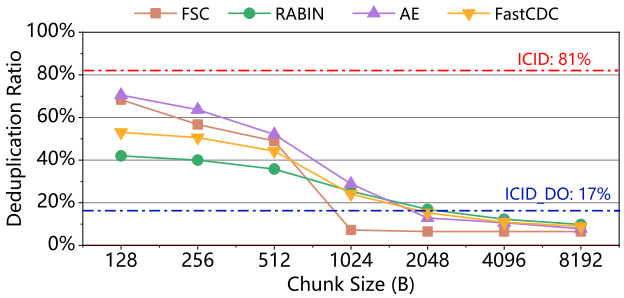
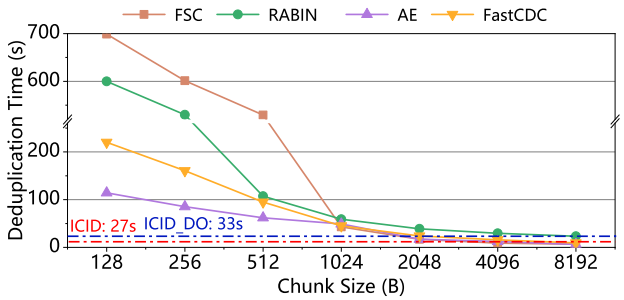


Fig. 13: Average time consumption of different steps in chunk-based approaches and ICID, all normalized to ICID.



(a) Deduplication ratio of LevelDB under different chunk sizes



(b) Deduplication time of LevelDB under different chunk sizes

Fig. 14: Deduplication ratio and time of LevelDB under different chunk sizes.

use a chunk size of 4 KB or 8 KB [13], [14] in practise. In contrast, ICID can always achieve high deduplication ratio no matter how the chunk size varies because ICID exploits the I/O causality between file operations rather than the content of data to identify data redundancy.

5.7 Memory Consumption

ICID needs to keep a rec-tree in memory to trace I/O-causality among in-memory file operations, and thus consumes an amount of memory. In order to demonstrate the necessity of our garbage collection scheme for memory-copy records, we implemented the third derivation of I/O-causality based deduplication scheme called ICID_KA. It does not reclaim outdated memory records and keeps all these records in memory. We use five datasets with different numbers of KV pairs, and the key and value sizes are 16 and 100 bytes, respectively. We measure the memory consumed by ICID, ICID_DO, and ICID_KA, FastCDC, respectively. As shown in Table 5, ICID consumes a small amount of memory no matter how many KV pairs are inserted into LevelDB, because ICID deletes outdated records timely. It even does not use up the reserved memory pool. The memory consumption of ICID_KA grows rapidly when the data

TABLE 5: Memory consumption of rec-tree for LevelDB

Schemes	Memory consumption (MB) for # million KVs				
	2	4	8	16	32
ICID	5.7	5.7	5.7	5.7	5.7
ICID_DO	0.1623	0.164	0.1643	0.1641	0.1644
ICID_KA	211.1	583.7	1542	3950	9894
FastCDC	1.1	2.4	4.2	9.3	18.3

TABLE 6: The ratio of metadata size to duplicated data size

Schemes	$\frac{metadata}{dedup.data} \times 100\%$ for # bytes of values						
	128	256	512	1024	2048	4096	8192
ICID	25.1	12.9	6.8	3.7	2.2	1.2	0.9
FastCDC	28.5	28.4	23.9	18.2	16.3	5.7	2.8

volume of KV pairs increase exponentially. Since ICID_KA keeps all memory-copy records in memory, it consumes up to 9 GB memory when the number of KV pairs becomes 32 millions. Because ICID_DO deletes all memory-copy records that overlap with the newly-inserted one, it significantly reduces the memory consumption to about 0.1 MB. However, this aggressive garbage collection strategy has a negative impact on the deduplication ratio, as shown in § 5.2. Chunk-based schemes use main memory to cache fingerprints of chunks, and the size of fingerprints grows linearly with the size of dataset. Since all chunk-based schemes split data into chunks in similar sizes, they generate almost the same amount of chunks and fingerprints. Therefore, we only take FastCDC as an example to present the memory consumption of chunk-based schemes in Table 5.

In summary, IDIC_DO shows rather low storage overhead, but results in low deduplication ratios. ICID_KA incurs significant storage overhead, and thus is impractical in real-world scenarios. ICID achieves high deduplication ratio at the expense of moderate storage overhead. As a result, ICID is the optimal choice in practice.

5.8 Storage Overhead of File Metadata

Although the data deduplication ratio is critical for a deduplication system, the metadata introduced by deduplication schemes may offset the saving of storage space. In this section, we focus on the storage overhead of metadata for both ICID and chunk-based deduplication systems (the rec-tree resides only in memory § 5.7). The metadata in ICID is mainly composed of the metadata used to index extents in ICID_FS. For chunk-based deduplication systems, the metadata includes both fingerprints and the data used to index chunks. We find that chunk-based deduplication systems show similar storage overhead of metadata, and thus only show the result of FastCDC in Table 6. For most value sizes, the ratio of the metadata size to the duplicated data size for ICID is much lower than that of chunk-based deduplication systems. The reason is that chunk-based systems cause additional storage overhead of fingerprints for all data, while ICID only needs to index the duplicate data.

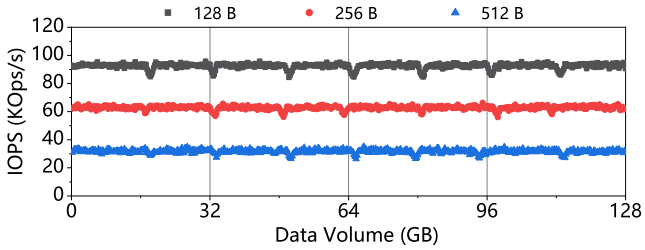


Fig. 15: The throughput of LevelDB with garbage collection

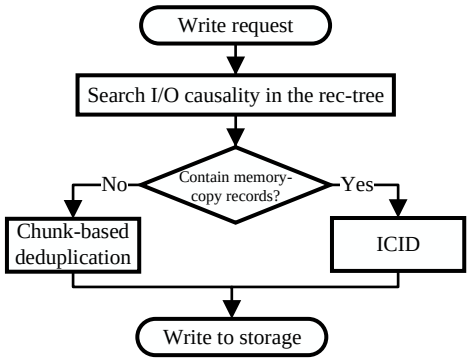


Fig. 16: Integration of chunk-based deduplication into ICID

5.9 Garbage Collection Performance

To evaluate the impact of GC on the throughput of LevelDB, we construct the data store with a 128 GB dataset, and update KV pairs continuously. The GC thread is manually triggered once the updated data volume exceeds 16 GB. Figure 15 illustrates how the IOPS varies with the data volume of write traffic. We can find that the performance degradation due to GC is very limited. For larger value sizes, the GC incurs less performance degradation because larger extents usually result in larger fragments, and thus less extents should be moved to generate large free space during the GC. Overall, GC has only a slight impact on the throughput of LevelDB.

6 Discussions

Integrating Chunk-based Deduplication into ICID. ICID is designed to deduplicate data redundancy caused by file operations (i.e., insert/merge/truncate). Because ICID offers high deduplication ratios and low overhead of deduplication, it can be an effective supplement to traditional chunk-based deduplication schemes. Thus, an optional deduplication framework that integrates chunk-based deduplication approaches into ICID can be applicable to different scenarios. As illustrated in Figure 16, for each write request of a process, the deduplication framework should first check whether there are correlated memory-copy records in the rec-tree. If no record is retrieved, the write request is forwarded to the chunk-based deduplication module. Otherwise, the write request is handled by ICID. Since the integrated deduplication framework does not change the I/O path of ICID, the performance of ICID remains unchanged. For the chunk-based deduplication module, the integrated deduplication framework only results in trivial performance overhead due to the lookup cost of the rec-tree. We note that only read and memory-copy operations would

generate new memory-copy records and update the rec-tree, while write operations only trigger a lookup of the rec-tree. Thus, if a program only receives data from periphery input devices and writes it to local storage, the rec-tree of this program would be always empty, incurring trivial performance overhead. Our experimental results show that the average latency for searching the rec-tree is 0.14 us. Compared with the 4.7 us latency of a write request (4 KB), the lookup cost of the rec-tree introduces only 3% additional latency, and the throughput remains unchanged.

Storage Fragmentation. Existing data deduplication systems face severe fragmentation problems. The fragmentation can cause random reads and lowers the throughput of data restoration. Some proposals exploit rewriting schemes to relocate unique chunks, and thus can mitigate the fragmentation problem to some extent [37]. A few proposals exploit other optimizations such as caching to accelerate the data restoration [38] due to fragmentation. However, the fragmentation problem is always accompanied with the data deduplication. We can explore defragmentation technologies [39] to mitigate this problem in an offline manner. On the other hand, since the random read performance of NVMs is much higher than that of SSDs/HDDs (Table 1), the fragmentation is no longer a critical problem for NVM-based storage systems.

Correctness of ICID. LevelDB maintains checksums of data blocks to verify data integrity. Thus, when a data block is read from storage, the accuracy of our deduplication scheme can be confirmed by validating the checksum. LevelDB would report an error once it finds a data block is corrupt. We perform identical computational tasks (including patching, compiling, and video editing) on ICID and a native system without deduplication. We compare the checksums of each file generated by these two systems. The consistent checksums also confirm the correctness of ICID implementation.

7 Conclusion

In this paper, we present an *I/O Causality-based In-line Deduplication* (ICID) scheme for NVM-based storage. ICID records the I/O causality among in-memory file operations to achieve in-line data deduplication, and avoid the time-consuming calculation of fingerprints caused by previous chunk-based deduplication schemes. We advocate two key technologies to manage memory-copy operations in a B-tree efficiently, i.e., a hybrid data structure to store memory-copy records, and a location-dependent garbage collection scheme for deleting outdated memory-copy records. Our experimental result demonstrates that ICID achieves up to $16\times$ higher deduplication ratio than state-of-the-art deduplication schemes, and also reduces the time overhead of data deduplication by 47% on average.

Acknowledgments

This work is supported jointly by National Key Research and Development Program of China under grant No.2022YFB4500303, and National Natural Science Foundation of China (NSFC) under grants No.62072198, 61825202, 61929103. This work is also sponsored by Huawei Technologies Co., Ltd (No. YBN2021035018A7).

References

[1] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, Inline Data Deduplication for Primary Storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, vol. 12, 2012, pp. 1–14.

[2] "Intel® Optane™ Technology," 2021, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[3] D. Yang, H. Liu, H. Jin, and Y. Zhang, "HMvisor: Dynamic Hybrid Memory Management for Virtual Machines," *Science China Information Sciences*, vol. 64, no. 9, pp. 192 104:1–16, 2021.

[4] Y. Xie, "Modeling, Architecture, and Applications for Emerging Memory Technologies," *IEEE Des. Test Comput.*, vol. 28, no. 1, pp. 44–51, 2011.

[5] C. Tian, H. Liu, X. Liao, and H. Jin, "UCat: Heterogeneous Memory Management for Unikernels," *Front. Comput. Sci.*, vol. 17, no. 1, pp. 171 204–171 215, 2022.

[6] 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 *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 14–23.

[7] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design Tradeoffs for Data Deduplication Performance in Backup Workloads," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 331–344.

[8] S. Mandal, G. Kuenning, D. Ok, V. Shastry, P. Shilane, S. Zhen, V. Tarasov, and E. Zadok, "Using Hints to Improve Inline Block-layer Deduplication," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 315–322.

[9] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, 2001.

[10] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A Comprehensive Study of the Past, Present, and Future of Data Deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[11] 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, 2014.

[12] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2011, pp. 25–39.

[13] C. Yu, C. Zhang, Y. Mao, and F. Li, "Leap-based Content Defined Chunking - Theory and Implementation," in *Proceedings of the IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–12.

[14] J. Park, J. Kim, Y. Kim, S. Lee, and O. Mutlu, "DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression," in *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, 2022, pp. 247–264.

[15] "libfuse," https://github.com/libfuse/libfuse.git.

[16] G. Fan, L. Yongkun, X. Yinlong, J. Song, and J. C. S. Lui, "SmartMD: A high performance deduplication engine with mixed pages," in *Proceedings of 2017 USENIX Annual Technical Conference (ATC)*, 2017, pp. 733–744.

[17] X. Nai, T. Chen, L. Yan, L. Hang, and W. Xiaoliang, "UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 325–340.

[18] 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 *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST)*, 2011, pp. 6–20.

[19] S. Quinlan and S. Dorward, "Venti: A New Approach to Archival Data Storage," in *Proceedings of the Conference on File and Storage Technologies (FAST)*, 2002, pp. 7–21.

[20] C. Wang, Q. Wei, J. Yang, C. Chen, Y. Yang, and M. Xue, "NV-Dedup: High-Performance Inline Deduplication for Non-Volatile Memory," *IEEE Trans. Comput.*, vol. 67, no. 5, pp. 658–671, 2017.

[21] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An Asymmetric Extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 1337–1345.

[22] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "FastCDC: a Fast and Efficient Content-Defined Chunking Approach for Data Deduplication," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016, pp. 101–114.
[23] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The Design of a Similarity Based Deduplication System," in *Proceedings of SYSTOR: The Israeli Experimental Systems Conference*, 2009, pp. 1–14.
[24] B. Zhu, K. Li, and H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 269–282.
[25] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," in *Proceedings of the 9th USENIX conference on File and stroage technologies (FAST)*, 2011, pp. 7–20.
[26] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok, "Dmdedup: Device mapper target for data deduplication," in *Proceedings of the 2014 Ottawa Linux Symposium (OLS)*, 2014, pp. 1–13.
[27] C. Ji, L.-P. Chang, R. Pan, C. Wu, C. Gao, L. Shi, T.-W. Kuo, and C. J. Xue, "Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021, pp. 127–140.
[28] M. Seo and S. Lim, "Deduplication flash file system with PRAM for non-linear editing," *IEEE Transactions on Consumer Electronics*, vol. 56, no. 3, pp. 1502–1510, 2010.
[29] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in Provenance Systems," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2009, pp. 10–24.
[30] "GNU Patch," https://savannah.gnu.org/projects/patch/.
[31] "FFmpeg," https://www.ffmpeg.org/.
[32] S. Ghemawat and J. Dean, "Google LevelDB," 2016, https://github.com/google/leveldb.
[33] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 273–286.
[34] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 323–338.
[35] "XXHash," https://github.com/Cyan4973/xxHash.git.
[36] A. Z. Broder, "Some applications of Rabin's fingerprinting method," in *Sequences II*, 1993, pp. 143–152.
[37] C. Ji, L.-P. Chang, R. Pan, C. Wu, C. Gao, L. Shi, T.-W. Kuo, and C. J. Xue, "Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2021, pp. 127–140.
[38] B. Mao, H. Jiang, S. Wu, Y. Fu, and L. Tian, "Read-Performance Optimization for Deduplication-Based Storage Systems in the Cloud," *ACM Trans. Storage*, vol. 10, no. 2, pp. 6:1–6:22, 2014.
[39] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "FFCCD: Fence-Free Crash-Consistent Concurrent Defragmentation for Persistent Memory," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022, pp. 274–288.

Haikun Liu is a professor in the School of Computer Science and Technology, Huazhong University of Science and Technology, China. He received his Ph.D degree in computer science and technology from HUST in 2012. He has co-authored more than 80 papers in prestigious conferences and Journals. His current research interests include in-memory computing, cloud computing, and distributed systems. He is a senior member of CCF, and a member of the IEEE.

Xiaozhong Jin received the bachelor's degree from North China Electric Power University, China, in 2019. He is currently working toward the PhD degree in Huazhong University of Science and Technology, China. His research interests include data deduplication and storage systems.

Chencheng Ye received Ph.D. in computer science from Huazhong University of Science and Technology, Wuhan, China, in 2019. He is currently an associate professor with the School of Computer Science and Technology, HUST. His research interests lie in hardware, and programming language supports for memory systems, with an emphasis on improving locality, programmability, and persistency.

Xiaofei Liao is a professor in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), China. He received his PhD degree in computer science and engineering from HUST, China, in 2005. He was awarded Excellent Youth Award from the National Science Foundation of China in 2018. His research interests are in the areas of computer architecture, system software, and big data processing. He is a member of IEEE and the IEEE Computer Society.

Hai Jin is a Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. Jin received his PhD in computer engineering from HUST in 1994. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is a Fellow of IEEE, a Fellow of CCF, and a life member of the ACM. He has co-authored more than 20 books and published over 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, and system security.

Yu Zhang received a Ph.D. degree in computer science from Huazhong University of Science and Technology (HUST) in 2016. He is now a associated professor in school of computer science and technology at HUST. His research interests include big data processing, graph computing and distributed systems. His current topic mainly focuses on application-driven big data processing and optimizations.