Light-Dedup: A Light-weight Inline Deduplication Framework for Non-Volatile Memory File Systems

Jiansheng Qiu [†][*][§], Yanqi Pan [†][*], Wen Xia[†]✉, Xiaojia Huang[†], Wenjun Wu[†], Xiangyu Zou[†], Shiyi Li[†], Yu Hua[‡]
[†]Harbin Institute of Technology, Shenzhen [‡]Huazhong University of Science and Technology
✉ Corresponding Author: Wen Xia (xiawen@hit.edu.cn)

Abstract

Emerging NVM is promising to become the next-generation storage media. However, its high cost hinders its development. Recent deduplication researches in NVM file systems demonstrate that NVM's cost can be reduced by eliminating redundant data blocks, but their design lacks complete insights into NVM's I/O mechanisms.

We propose Light-Dedup, a light-weight inline deduplication framework for NVM file systems that performs fast block-level deduplication while taking NVM's I/O mechanisms into consideration. Specifically, Light-Dedup proposes Light-Redundant-Block-Identifier (LRBI), which combines non-cryptographic hash with a speculative-prefetch-based byte-by-byte content-comparison approach. LRBI leverages the memory interface of NVM to enable asynchronous reads by speculatively prefetching in-NVM data blocks into the CPU/NVM buffers. Thus, NVM's read latency seen by content-comparison is markedly reduced due to buffer hits. Moreover, Light-Dedup adopts an in-NVM Light-Meta-Table (LMT) to store deduplication metadata and collaborate with LRBI. LMT is organized in the *region* granularity, which significantly reduces metadata I/O amplification and improves deduplication performance.

Experimental results suggest Light-Dedup achieves 1.01–8.98$\times$ I/O throughput over the state-of-the-art NVM deduplication file systems. Here, the speculative prefetch technique used in LRBI improves Light-Dedup by 0.3–118%. In addition, the region-based layout of LMT reduces metadata read/write amplification from 19.35$\times$/9.86$\times$ to 6.10$\times$/3.43$\times$ in our hand-crafted aging workload.

1 Introduction

Recently, Non-Volatile Memory (NVM) has been becoming increasingly popular. Its byte-addressability, persistence, and low latency enable it to be attached to the memory bus, sitting alongside the DRAM [24, 44, 52, 67]. Optane DC Persistent Memory Module (DCPMM) is the latest commercially available NVM. However, it is much more expensive than Hard Disk Drive (HDD) and Solid State Drive (SSD). Therefore, reducing the price of NVM is paramount for its future usage.

Deduplication, a system-level data compression approach, can enlarge the logical space and reduce the amortized cost of storage devices [16, 56, 62, 68]. Deduplication is widely used in file systems [72], backup systems [18–20, 39, 63], cloud computing [37, 57], etc. It usually calculates the fingerprints of data blocks and then identifies duplicates according to their fingerprints. For the redundant block, deduplication increments the reference count in the corresponding metadata to maintain data integrity.

Traditional disk-based deduplication approaches, such as using the cryptographic hash (e.g., SHA-256 and MD-5) to identify redundant data blocks, do not fit well with NVM since fast NVM has shifted the performance bottleneck from I/O to CPU. Prior works on deduplication for NVM [7, 28, 58, 75] propose several ways to address the issues. First, some works use offline deduplication to reduce the overhead of deduplication on the critical path, such as DeNOVA [28]. However, such background deduplication can neither enhance the file systems' write performance nor improve NVM's endurance. Second, many works use the non-cryptographic hash (e.g., CRC32 and xxHash [11]) to accelerate the identification of duplicate blocks. For example, NV-Dedup [58] leverages non-cryptographic hash to avoid calculating cryptographic hashes for most unique blocks, while DeWrite [75] shows that combing the non-cryptographic hash with byte-by-byte comparison is efficient for deduplication at cache line granularity.

Despite these efforts, existing works still fail to fully exploit the performance of NVM during deduplication due to a lack of comprehensive insights into NVM's I/O mechanisms. First, NVM's read/write asymmetry encourages researchers to combine non-cryptographic hash with byte-by-byte content-comparison to quickly identify the duplicate data [67, 75]. Thus the overheads of cryptographic hash calculation can be eliminated. Second, we observe other two NVM I/O features that hinder NVM deduplication performance: (1) *Long media read latency*. Despite its read/write asymmetry, NVM's read latency is 2–3$\times$ higher than the write since the write buffer inside NVM hides the long media write latency [67]. Therefore, there is still a large room left for the acceleration of content-comparison by hiding the read latency. (2) *Coarse media access granularity*. The mismatch between the size of deduplication metadata (commonly 16–64 bytes for each data block) and the coarse media access granularity in NVM (e.g., 256 bytes XPLine of DCPMM) can lead to severe metadata I/O amplification if the access to the metadata lacks locality,

[*]Jiansheng Qiu and Yanqi Pan are co-first authors of the paper.
[§]Now working at Tsinghua University.

which degrades not only deduplication performance but also NVM's endurance, especially when the system is aged.

This paper presents Light-Dedup, a novel light-weight inline deduplication framework for NVM file systems. Light-Dedup is designed with two specific goals in mind: (1) Maximizing the deduplication performance by considering NVM's memory interface, read/write asymmetry, and access granularity, while adding negligible overhead to the critical path. (2) Retaining low deduplication metadata I/O amplification even if the file system is severely aged (i.e., many holes).

To achieve the first goal, Light-Dedup proposes Light-Redundant-Block-Identifier (LRBI) to quickly identify the duplicate blocks. Unlike the prior works that use both non-cryptographic and cryptographic hash [58] or that straightforwardly combine non-cryptographic hash with byte-by-byte content-comparison [75], LRBI considers both NVM's read/write asymmetry and long media read latency in redundant block identification. Specifically, LRBI uses xxHash, one of the fastest non-cryptographic hashes [11], to quickly identify most non-duplicate blocks. For those blocks with the same fingerprint, LRBI leverages NVM's memory interface to enable asynchronous NVM reads and proposes speculative prefetch to minimize the read latency seen by content-comparison. In particular, speculative prefetch uses *In-Block* and *Cross-Block Prefetch* to exploit the parallelism between NVM read and CPU computation.

To achieve the second goal, Light-Dedup organizes its in-NVM deduplication metadata table, Light-Meta-Table (LMT), as a *region*-based linked list. Each *region* contains multiple continuous metadata entries. Each entry stores the critical information for both basic deduplication and speculative prefetch used in LRBI. The allocation of metadata entries is done first by allocating a *region* and then by allocating entries in that region almost sequentially, which significantly reduces the deduplication metadata I/O amplification caused by NVM's coarse access granularity, especially in an aged file system. In addition, LMT trades $1\times$ extra deduplication metadata space usage for zero garbage collection overheads, which retains the stabilization of deduplication performance.

In summary, this paper makes the following contributions:

- We perform an in-depth analysis of how deduplication can be affected by several NVM's I/O mechanisms and introduce how to maximize NVM deduplication performance with full consideration of them.
- We propose an inline deduplication framework for NVM file systems, Light-Dedup, with two key techniques: (1) LRBI combines non-cryptographic hash with speculative-prefetch-based content-comparison to fully leverage NVM's I/O asymmetry while hiding its media read latency by enabling asynchronous NVM reads. (2) The *region*-based layout is adopted in LMT to manage deduplication metadata with a good locality and retain low metadata I/O amplification.
- We implement Light-Dedup in Linux kernel 5.1.0 based on NOVA [66], one state-of-the-art NVM file system. The code is available at https://github.com/Light-Dedup/Light-Dedup. Furthermore, we make a comprehensive evaluation of various synthetic and real-world workloads. The results show that Light-Dedup adds negligible overhead while significantly improving the file system's write performance under a high duplication ratio.

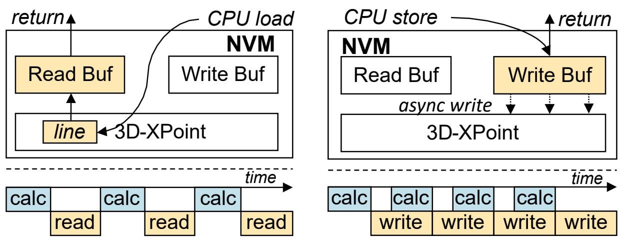
2 Background and Related Work

2.1 NVM and NVM File Systems

With its byte-addressability, low latency, persistence, and low power consumption [3, 21, 26, 35, 40, 41, 46], NVM becomes a promising candidate for next-generation storage media. In this work, we focus on high-density storage-type NVM with the memory-like interface [29] that serves as persistent storage media. For brevity, we denote such storage-type NVM as NVM. According to the latest research on the commercial DCPMM [64, 67] and our investigation on existing NVM devices [6, 34, 50, 51, 65, 71], this paper concludes the following **five common I/O features** of NVM that potentially have impacts on NVM deduplication performance:

- *Asymmetry in Read/Write Bandwith*. The read bandwidth of NVM is up to $3\times$ than its write [67]. The feature is common for persistent storage media such as Phase Change Memory (PCM) [50, 71], STT-RAM [6, 34], memristor [65], 3D-XPoint [67], NAND flash [1], etc.
- *I/O with Buffers*. For writes, NVM leverages the buffer to write asynchronously to hide long media write latency. While for uncached reads, NVM fetches data synchronously from the media. The data will be cached in the internal read buffer for future reads [51, 64, 67]. Figure 1 shows the I/O mechanisms of NVM.
- *Coarse Access Granularity*. Coarse media access granularity is common for storage-type NVM. For example, the row buffer size of a PCM is preferred to be larger than 128 bytes [30]. Coarse access granularity (and the above I/O buffers) is beneficial for improving storage bandwidth and bridging the performance gap between storage and CPU. Since NVM is denser but slower compared to DRAM, it is reasonable that NVM has a larger access granularity than a cache line.
- *Long Media Read Latency*. The underlying non-volatile media generally introduces relatively longer media latency than DRAM [29, 30, 43, 54]. The synchronous data fetch mechanism fails to hide such latency [64].
- *Memory Interface*. With the memory interface, NVM can be accessed by CPU store/load. This feature makes asynchronous CPU prefetch possible, which can be leveraged to address NVM's long media read latency.

To well exploit the physical characteristics of NVM devices, several NVM file systems [8, 12, 17, 70] are proposed. Among these file systems, NOVA [66] is the state-of-the-art one, which aims to exploit the potential of DRAM and NVM



(a) Synchronous reads (b) Asynchronous writes

Figure 1: NVM I/O buffering mechanisms.

hybrid memory systems while providing strong consistency guarantees. Specifically, NOVA allocates a separate log for each inode (i.e., file), appends single inode operations as entries of the inode's log, and atomically updates the log's tail to commit the operations. For operations involving multiple inodes, NOVA records the log tail's pointers of the affected inodes in the journal to update them atomically. Further, NOVA accelerates search operations by maintaining radix trees of directories and files in DRAM.

2.2 Inline Deduplication Techniques for NVM

File system deduplication [62] is a block-level redundancy elimination technique and has been needed in many applications [18–20, 59, 63, 72]. This paper focuses on inline deduplication since the offline approach neither enhances file systems' write performance nor improves NVM's endurance. Typically, an inline deduplication framework consists of several techniques, including redundancy identification, deduplication metadata management, indexing, etc.

Redundant Block Identification Techniques. Traditional disk-based deduplication approaches, such as using the cryptographic hash to determine the duplicates [16, 56, 68], do not fit well with fast NVM devices due to their heavy software overhead. Recent works [7, 58, 75] leverage non-cryptographic hash functions to reduce the computation cost of fingerprints. However, non-cryptographic hash suffers from hash collision (different blocks have the same hash). To address the issue, existing approaches either apply cryptographic hash (e.g., NV-Dedup [58]) or straightforwardly perform byte-by-byte content-comparison (e.g., DeWrite [75]) to verify if the blocks are duplicate when their non-cryptographic hashes equal[1]. However, (1) cryptographic hash calculation is a bottleneck when there are many duplicates; (2) the long latency of uncached reads hinders the performance of content-comparison, especially when the concurrency level is low and read/write asymmetry is not obvious. In contrast, our approach (i.e., LRBI) combines non-cryptographic hash with a speculative-prefetch-based content-comparison technique to exploit I/O asymmetry and hide long media read latency.

In-Storage Deduplication Metadata Management. To manage redundant blocks, deduplication approaches must maintain in-storage structures to store the basic information about the deduplicated blocks (e.g., the mappings between fingerprints and physical blocks, reference count, etc.). Besides, some additional bits are required for the collaboration with redundancy identification. Existing NVM deduplication approaches (e.g., NV-Dedup [58], LO-Dedup [7], DeWrite [75]) often reserve a fixed in-NVM table to store the deduplication metadata and to allocate/free them by the free list. Additionally, NV-Dedup maintains both non-cryptographic and cryptographic hash in the table; LO-Dedup organizes the deduplication metadata as an ordered linked list structure to accelerate the continuous matching. However, such free-list-based management is not NVM-friendly since its allocation strategy can introduce significant fragmentation when the system is aged, causing a severe metadata I/O amplification. In contrast, our proposed LMT manages the deduplication metadata as a *region*-based linked list, provides course-grained metadata management, and significantly reduces metadata I/O amplification under aged file systems.

In-NVM Deduplication Metadata Index Techniques. Searching for in-NVM deduplication metadata based on the calculated fingerprint is a critical step in deduplication. To prevent frequent NVM accesses, existing works usually build an in-DRAM index, such as static hash tables or red-black tree, to accelerate the search [7, 58]. We use a dynamic hash table (i.e., *rhashtable* [13]) for its resizability and efficiency.

3 Observations and Motivations

3.1 Data Redundancy & NVM Deduplication

Data redundancy is a common phenomenon in storage systems with the exponential growth of data. Prior works [19, 20, 63, 72] have observed a large number of redundancies in modern primary storage systems. For example, there are 95% and 47% duplicates (in 4 KiB block granularity) in two real-world traces collected by FIU: *Mails* and *WebVMs* [27, 33]. Thus, the deduplication approach is a promising solution to enlarge logical storage space and reduce storage costs. As the next-generation storage media, deduplication for expensive NVM is profitable and urgent. Recently, many research efforts have designed deduplication schemes specifically for NVM file systems [7, 58]. However, they fail to fully exploit the characteristics of NVM's I/O mechanisms and leave substantial room for improvement from the performance point of view. This paper aims to develop a more efficient inline deduplication framework scheme for NVM file systems that can fully exploit the I/O characteristics of NVM devices.

3.2 I/O Asymmetry and Read Latency in NVM Redundant Block Identification

The efficiency of redundant block identification is essential to NVM deduplication performance. Traditional disk-based deduplication approaches use the cryptographic hash to identify the duplicate blocks [16, 32, 56, 68]. However, it does not suit NVM deduplication well due to its computation overhead, which wastes much CPU computation and thus starves

[1]LO-Dedup does not address the hash collision of the non-cryptographic hash used in their paper. Thus, we omit its redundancy identification technique.

Table 1: The breakdown deduplication time. Light denotes Light-Dedup, and *LD-w/o-P* denotes a simple deduplication file system that incorporates non-cryptographic hash with content-comparison into the write path and deduplicates 4 KiB blocks. With the introduced speculative prefetch technique (i.e., Light), content-comparison time is dropped by 62.2%.

System	Calc. Lat (ns)		I/O Lat (ns)		Bandwidth
	fp	others	write	cmp	(MiB/s)
NOVA	0.0	84.7	2275.6	0.0	1401
LD-w/o-P (1st)	309.9	1072.5	**585.3**	0.0	1612
LD-w/o-P (2nd)	308.0	571.6	0.0	**3263.0**	870
Light (1st)	310.0	1131.3	**559.8**	0.0	1592
Light (2nd)	0.0	343.3	0.0	**1234.8**	1914

NVM. We observe the problem in NV-Dedup [58]. In a simple sequential write 4 GiB workload, the write bandwidth of NV-Dedup drops by 52.5% with the duplication ratio increasing from 0% to 75%. The root cause is that cryptographic hash calculation (i.e., MD-5) dominates up to 64.9% of the whole write time since NV-Dedup relies on the cryptographic hash to handle its non-cryptographic hash collision.

To address the above heavy computation, DeWrite [75] uses non-cryptographic hash and byte-by-byte comparison in the combined manner [10]. The method is well aligned with the characteristic of NVM since it prevents heavy CPU computation and leverages the large read/write asymmetry to trade slow duplicate writes for faster reads (i.e., content-comparison). However, recent researches about NVM reads [64, 67] show that they can still be a bottleneck since uncached reads have to fetch data from media synchronously, which introduces long media read latency and thus negatively affects the content-comparison performance.

We examine how exactly NVM I/O affects the deduplication performance. Table 1 shows the breakdown latency of orderly writing two 4 GiB files with identical content (2 MiB per I/O) to NOVA and *LD-w/o-P*[2] under a single thread. Note that the first write conducts no duplicate blocks, but the second write causes 100% duplicates and results in reads (caused by content-comparison). We observe several interesting phenomena from Table 1. First, *LD-w/o-P* has a surprisingly low write time (585.3ms) compared to NOVA (2275.6ms) because asynchronous write enables the parallelism between CPU computation and write I/O. Thus, computation hides part of NVM writes latency (as shown in Figure 1b). Second, during the second writes, the write bandwidth of *LD-w/o-P* drops by 46% compared to the first. We find that the content-comparison time arises to 3263.0ns, which dominates 78.8% deduplication latency. The above observations suggest that non-cryptographic hash-based redundant block identification adds negligible overheads to the normal non-deduplication

[2]We build *LD-w/o-P* based on our proposed Light-Dedup by removing the speculative prefetch technique. This means that the deduplication metadata management and indexing are the same as in Light-Dedup, but they have negligible performance impacts on the experiments of this subsection.

Table 2: The average NVM extra reads/writes of deduplication metadata for writing each block.

Approaches	First Write		Second Write	
	Read (B)	Write (B)	Read (B)	Write (B)
ideal	≈40	40	40	≈40
All-in-NVM	726.12	293.17	528.65	259.05
Entry-based	126.94	79.56	774.13	394.54
Ours	116.28	75.75	244.19	137.17

write path (i.e., the data blocks to be written are all unique). However, deduplication performance is significantly limited by the long read latency and is far from ideal. We believe there are two reasons: (1) NVM's read/write asymmetry under low thread count is not large enough [67]. (2) Current hardware prefetcher of intel 64 bits architecture fails to remedy the drawbacks of NVM's long media read latency since it is designed for DRAM and only attempts to prefetch two cache lines ahead of the prefetch stream [23].

In summary, NVM's long read latency hinders content-comparison performance during NVM deduplication. Considering NVM's asynchronous writes and the limitations of hardware prefetcher, we are motivated to think: Can we manually achieve asynchronous reads to hide media read latency? Memory characteristics of NVM inspire us to obtain our first motivation: We can leverage memory prefetch instructions to enable asynchronous NVM reads and thus accelerate content-comparsion. However, applying prefetch to NVM deduplication is not straightforward. There are two technical challenges: (1) The limited number of concurrent prefetch instructions that a CPU core can handle. (2) How to incorporate the prefetch mechanism into deduplication logic.

3.3 Metadata I/O Amplification in NVM Deduplication Metadata Management

During NVM deduplication, deduplication metadata can be frequently accessed and updated, causing a large amount of small NVM accesses. Metadata I/O amplification will get larger if these small NVM accesses exhibit a random pattern due to NVM's coarse access granularity. However, existing NVM deduplication file systems pay little attention to the issue. In this subsection, we investigate two widely used NVM deduplication metadata management approaches.

All-in-NVM Management. DeNOVA [28] takes an All-in-NVM design and constructs an in-NVM hash table to store and index the deduplication metadata to reduce DRAM consumption. However, hash tables typically exhibit random access patterns [36, 45, 64, 73, 74], which leads to severe read/write amplification. To verify this, we implement an inline deduplication system with an All-in-NVM design based on *Light-Dedup*: Its deduplication metadata are organized as a static hash table in NVM. We write the same 64 GiB file twice and measure the extra NVM reads/writes of deduplication metadata using *ipmctl* [22]. The experimental results are shown in Table 2. Ideally, each block write results in about 40 bytes NVM read/write (i.e., 32 bytes for deduplication metadata entry, 8 bytes for the mapping from block number to metadata entry, see §4.2). In this case, read and write amplification are $528.65/40 \approx 13.2\times$ and $293.17/40 \approx 7.3\times$. Note that the reason why read amplification is nearly $2\times$ of theoretical upper bound ($8\times$) may be due to the prefetch mechanism and buffering strategy of internal Optane DCPMM hardware [64].

NVM-DRAM Hybrid Entry-Based Management. NV-Dedup [58] and LO-Dedup [7] store the deduplication metadata in NVM and maintain the in-DRAM index to locate them efficiently, which alleviates the problem of all-in-NVM design. Their deduplication metadata is managed at the granularity of cache lines, aligned in a manner favored by CPUs, and allocated/freed through a free list. We refer to this approach as *entry-based*. It is acceptable for a fresh new file system, but when the file system is aged, the physical location of allocated free entries can be random, which results in random access to NVM and causes severe read/write amplification. To show the problem, we make an extensive evaluation in §5.5 and present part of the results in the *Entry-based* row in Table 2, in which the first write is performed in the fresh new system and the second write is in the aged system. The results suggest that such entry-based metadata management can lead to significant read/write amplification in the aged system, about $774.13/40 \approx 19.35\times$ and $394.54/40 \approx 9.86\times$, respectively.

In summary, the severe metadata I/O amplification observed in Table 2 wears out NVMs and leads to performance degradation under aging file systems. To alleviate the problem, we focus on redesigning the hybrid deduplication metadata management strategy. Inspired by mimalloc [31], which shards its free list in page granularity, we obtain our second motivation: managing deduplication metadata in the *region* (i.e., 4 KiB block) granularity to maintain access locality, which elegantly reduces metadata I/O amplification. However, the issue of how to reclaim stale entries (i.e., garbage collection) with minimal overhead and design entry fields that collaborate with LRBI remains unresolved.

4 Design and Implementation

4.1 System Overview

Based on the observations of NVM's internal I/O mechanisms, we propose Light-Dedup, a light-weight inline deduplication framework for NVM file systems, as shown in Figure 2. It includes two key techniques:

- **Light-Redundant-Block-Identifier (LRBI)**. LRBI is proposed to quickly identify duplicate blocks by exploiting NVM's large read/write asymmetry and hiding long media read latency. It combines non-cryptographic hash with a speculative-prefetch-based byte-by-byte content-comparison technique. Specifically, speculative prefetch leverages NVM's memory interface and uses *In-Block* and *Cross-Block Prefetch* techniques to asynchronously load speculated data into CPU/NVM buffers, which exploits the parallelism of NVM I/O and CPU computation and thus markedly hides read latency.

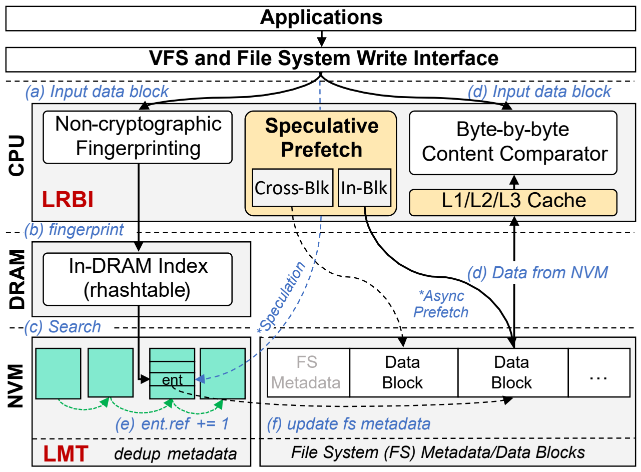


Figure 2: Light-Dedup overview.

- **Light-Meta-Table (LMT)**. LMT is an in-NVM table responsible for (1) storing basic deduplication metadata, such as the mapping from fingerprints to physical blocks; (2) maintaining speculation information, such as the hint of where and whether to prefetch the to-be-compared block. LMT adopts *region-based layout* to retain the locality of deduplication metadata. A *region* is a 4 KiB block, and *regions* are linked by 8 bytes pointers. Each dedup metadata is allocated in the *region* almost sequentially, hence reducing metadata I/O amplification. In addition, to prevent GC overheads brought by sequentiality, LMT trades $1\times$ more space for zero GC overheads.

In-DRAM Index. Light-Dedup adapts a *rhashtable* in DRAM to locate in-NVM deduplication metadata entry, whose key is the hash value (i.e., fingerprint) and value is the in-NVM position of the corresponding entry. We use *rhashtable* since it is a mature, well-tested, and efficient dynamic hash table implementation in the mainline Linux kernel [13], which suits for point-query (i.e., searching for a specific deduplication metadata entry by the given fingerprint). Meanwhile, *rhashtable* leverages efficient Read-Copy-Update (RCU) Lock [38] to handle the concurrent accesses to LMT. Note that this paper does not aim to redesign a specific index structure since in-DRAM hash table indexing is commonly efficient (e.g., usually achieves constant access time), and there have been many works dedicated on this [36, 45, 73, 74].

Put NVM I/O Features & Light-Dedup Together. We summarize Light-Dedup's insights into the presented NVM I/O features. First, read/write asymmetry can be leveraged to improve NVM deduplication performance by combing non-cryptographic hash with content-comparison, trading the slow duplicate writes for the faster reads (§4.2). Second, this paper observes that long media read latency hinders content-comparison performance dramatically. To address the problem, we propose LRBI, which follows the non-cryptographic hash-based infrastructure but leverages NVM's memory interface and I/O buffering to enable asynchronous reads, and thus markedly accelerates content-comparison (§4.3). Third, Light-Dedup addresses the metadata I/O amplification brought by NVM's coarse access granularity by organizing LMT as a *region*-based linked list (§4.4).

4.2 Basic Deduplication Logic

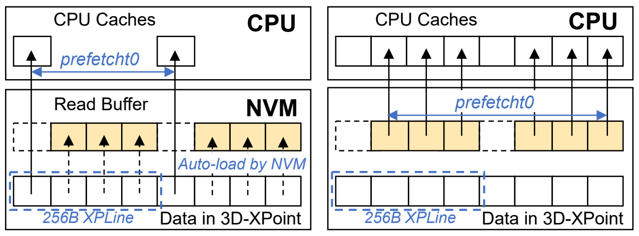
This subsection will introduce the basic deduplication flows of Light-Dedup (without speculative prefetch) to show how non-cryptographic hash and byte-by-byte comparison can be integrated into NVM file systems. Notably, we refer to this version as *LD-w/o-P*, as mentioned in §3.2.

Write Logic. Assume that the file system is writing a 4 KiB duplicate block. As shown in Figure 2, during the write, (1) Light-Dedup first calculates the non-cryptographic hash as the fingerprint of the input block (see steps (a), (b)), (2) and then efficiently locates the corresponding deduplication metadata entry (refer to as entry for brevity) in LMT by searching for the given fingerprint in *rhashtable* (see step (c)). (3) Once the entry is found (i.e., hash value matches), Light-Dedup compares the content of the input block and the block corresponding to the found entry byte-by-byte (see step (d)). (4) Finally, assuming comparison determines that the input block is duplicate, Light-Dedup increments the reference count of the duplicate block and records its duplicate block number in the file system metadata (see steps (e), (f)).

There are two exceptional cases to be handled. Given the fingerprint of the input block, (1) if no entry with the same fingerprint is found in LMT, suggesting that the input block is unique, then it will be written normally by the file system. After that, both its fingerprint and the block number are stored in a newly allocated entry with the reference count set to one. (2) If an entry is found, but the content of the stored block and the input one are different, then the input block becomes a non-dedup block, i.e., Light-Dedup does not allocate an entry for it, which will not affect the correctness of the deduplication system since only the file system has access to that block.

Deletion Logic. Light-Dedup maintains another in-NVM table that maps the block number to the offset of the corresponding deduplication metadata entry (i.e., similar to an inverse index). Note that the mapping maintenance introduces another 8 Byte metadata write in the write path (this is why the ideal NVM access is 40 bytes instead of 32 bytes). To delete a block, Light-Dedup first locates its deduplication metadata entry with the aforementioned table. If the entry is not found in the table, suggesting that the block number has not been inserted into the metadata table, then Light-Dedup frees the block directly. Otherwise, Light-Dedup decreases the reference count of the block (recorded in the entry) by one. If the reference count becomes zero, then the block can be safely freed. Deleting blocks in Light-Dedup does not cause the garbage collection of deduplication metadata due to the tradeoff in LMT (see §4.4).

Read Logic. The read path remains the same as the non-dedup file systems. Note that deduplication fragments files' data, which may lead to random reads. However, Light-Dedup deduplicates 4 KiB blocks and large random access to NVM does not cause significant performance degradation [64].



(a) 1st: Prefetching XPLines. (b) 2nd: Prefetching read buffer.

Figure 3: *In-Block Prefetch (IBP)* mechanisms. Each square in the figure indicates one 64 bytes cache line.

4.3 LRBI: Dedup with Speculative Prefetch

In this subsection, we present the step-by-step design of speculative prefetch used in LRBI, which aims to reduce the read latency seen by content-comparison (§3.2). Its key design principle is to enable asynchronous NVM reads and to markedly improve the parallelism between NVM I/O and CPU computation. It consists of two prefetch strategies: *In-Block Prefetch* and *Cross-Block Prefetch*.

4.3.1 In-Block Prefetch (IBP)

IBP speculates that the content-comparison always tends to compare all bytes and leverages memory prefetch instructions, e.g., `prefetcht0` assembly instruction in x86 [23], to enhance the parallelism of reading bytes in the same block.

Prefetch-Cmp-64 (P64). The most straightforward prefetch strategy is issuing 64 prefetch instructions to load 64 cache lines of the block into CPU caches (i.e., $64 \times 64 = 4096$ bytes) before comparing it with the input block (if they have the same fingerprint value). However, the maximum number of concurrent prefetch instructions a CPU core can handle is limited. In our machine, that number is in the open range $(8, 16)$. Therefore, many prefetch instructions in *P64* are executed in a serial manner, and thus the parallelism is limited.

In-Block Prefetch (IBP). We leverage the large access granularity of NVM (e.g., 256 bytes XPLine) to address the problem of *P64*. As Figure 3 shows, first, we issue 16 prefetch instructions with stride 256 bytes to touch the first cache line of XPLine, and the whole block is loaded automatically into CPU caches or NVM read buffer [64], where most of the NVM read is in parallel. Second, we issue prefetch instructions with stride 64 bytes to bring the in-read-buffer data into CPU caches. Note that *In-Block Prefetch* is a general optimization technique that can be applied to some other block-based NVM I/O scenarios, but it is beyond the scope of this paper.

4.3.2 Cross-Block Prefetch (CBP)

Unlike *IBP*, *Cross-Block Prefetch* exploits the parallelism among CPU tasks (i.e., fingerprint calculations and content-comparison, etc.) and NVM I/O tasks (i.e., reading a to-be-compared data block), and thus hides NVM's media read

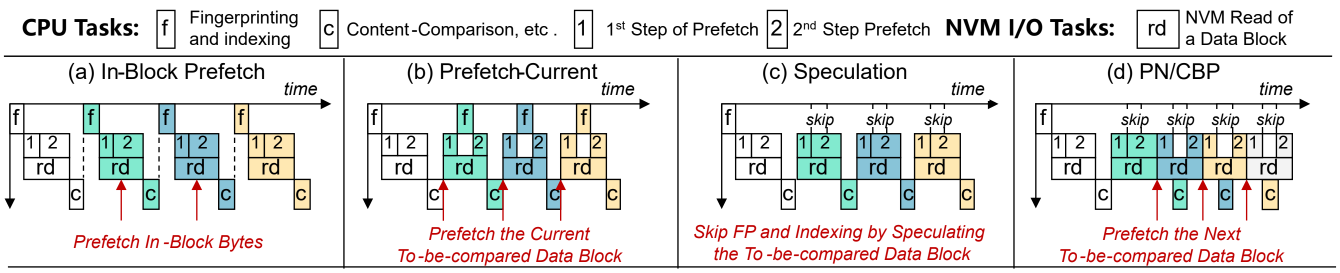

CPU Tasks:
f Fingerprinting and indexing
c Content-Comparison, etc.
1 1st Step of Prefetch
2 2nd Step Prefetch
NVM I/O Tasks:
rd NVM Read of a Data Block
(a) In-Block Prefetch
(b) Prefetch-Current
(c) Speculation
(d) PN/CBP
time
skip
Prefetch In-Block Bytes
Prefetch the Current To-be-compared Data Block
Skip FP and Indexing by Speculating the To-be-compared Data Block
Prefetch the Next To-be-compared Data Block

Figure 4: Simplified time sequence of Light-Dedup using different prefetch mechanisms: *IBP* improves parallelism of reading an NVM block while *PC*/*SP*/*PN*/*CBP* further improves the parallelism between CPU and NVM I/O tasks. Here, task $\boxed{c}$ includes content-comparison, updating dedup/FS metadata, etc. Note that the time of running task $\boxed{2}$ depends on the amount of data loaded into NVM's read buffer. Thus, task $\boxed{2}$ can be faster (shorter) after running tasks $\boxed{f}$ and $\boxed{c}$ in *PC* and *PN*/*CBP*, respectively.

latency. This subsection introduces the technique with step-by-step explorations, as shown in Figures 4(b)–(d).

Determining Where and Whether to Prefetch the To-be-compared Data Block. This is critical for Light-Dedup to support *Cross-Block Prefetch*. First, we incorporate a "hint" field into the deduplication metadata entry. The hint maintains both the location of the subsequent speculated block's entry and the *trust degree* (range: $[0,7]$) that indicates if the speculation is correct. Second, to maintain the hint, we keep track of the entry of the last written block and check the corresponding hint after the deduplication of the current input block finishes: *trust degree* is decreased by 2 if the hint is proven to be wrong, and increased by 1 if correct. The hint is trusted and followed only if the trust degree reaches the trust threshold (i.e., 4 for now). Third, since prefetch consumes NVM read bandwidth, we are conservative about its usage. Thus, we further introduce *per-CPU stream trust degree* to indicate the locality of the workload, which is increased/decreased along with per-entry *trust degree*s. Prefetch is enabled only if the *stream trust degree* of the current CPU reaches its maximum value (i.e., 7). Removing *trust degree* reduces the hit rate of CBP from 98.6% to 60.0% during WebVMs batch replay (the details of the workload are presented in §5.3), which shows the effectiveness of *trust degrees*.

Prefetch-Current (PC). As Figure 4(b) shows, the simplest idea is to prefetch the stored block that is possible to be compared with the current input block (based on the stored hints) before any deduplication calculations, and then follow the basic deduplication logic to deduplicate the input block. In this way, fingerprint calculations and index looking up ($\boxed{f}$ in Figure 4) are executed parallelly with NVM reads, and then part of the NVM read latency can be hidden.

Speculation (SP). Furthermore, as Figure 4(c) shows, we leverage hints to skip the fingerprint calculations and indexing by directly locating the deduplication metadata entry (see **Speculation* arrow in Figure 2). If the content-comparison demonstrates that the compared two blocks are the same, then the duplicate block is found; otherwise, we fall back to the basic deduplication logic. *SP* outperforms *PC* since *SP* can reduce (skip) many deduplication calculations.

Prefetch-Next (PN). More aggressively, as Figure 4(d) shows, to maximize parallelism, we utilize the stored speculation hints to suggest the block that is likely to be compared with the subsequent input block, and then we prefetch that block after loading the current to-be-compared block into CPU caches ($\boxed{2}$ in Figure 4). Therefore, the NVM read of the next to-be-compared can be parallel with the following CPU tasks (e.g., content-comparison). Now, NVM read is almost fully parallel with the CPU computations.

Cross-Block Prefetch (CBP). From the above explorations, we take *PN* as the fundamental of cross-block prefetch. However, we find that *PN* significantly degrades the deduplication performance at a high concurrency level (as shown in Figure 11 later in §5.4) since the large amount of extra prefetch I/O exacerbates the contention of NVM read buffer. Thus, we further introduce *Transition* technique to mitigate the issue by dynamically enabling/disabling prefetch according to concurrency level. Specifically, we maintain the number of threads that access NVM concurrently with an atomic variable and do not prefetch the next block if the number of threads reaches the specified threshold. Note that the threshold is a kernel module parameter. It is set to 6 by default (and we use this default value in our tests) because according to Figure 11, PN's throughput drops below SP's throughput when the number of threads ≥ 6 due to buffer contentions. Now, we obtain the final version of *CBP* (i.e., *PN*+*Transition*).

4.3.3 Speculative Prefetch: Put IBP and CBP Together

Generally, speculative prefetch enables the asynchronous NVM reads for content-comparison at both byte and block levels. Among them, CBP is frequently triggered when the workload exhibits good duplication continuousness (i.e., most hints are trusted according to trust degrees). Otherwise, Light-Dedup falls back to IBP when the duplication continuousness is poor (since most hints are not trusted). Therefore, the functionalities of CBP and IBP are complementary and are combined together to deliver fast content-comparison performance in both cases. The performance evaluation shows that speculative prefetch can achieve up to 118% performance improvement. More details can be obtained in §5.4.

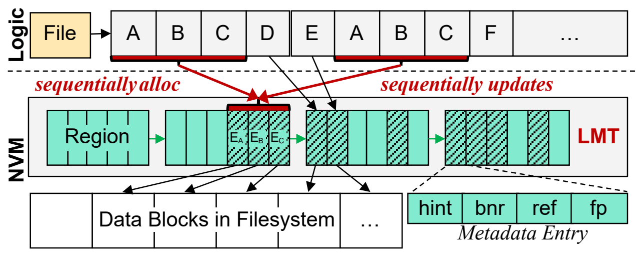


Figure 5: Illustration of Light-Meta-Table (LMT).

4.4 LMT: In-NVM Dedup Metadata Layout

In this subsection, we present the design of LMT, as illustrated in Figure 5. LMT is used for maintaining the mappings from fingerprints to physical blocks and providing hints for speculative prefetch. Furthermore, the deduplication metadata in LMT are laid out in the *region* granularity to reduce the read/write amplification of deduplication metadata access in NVM (i.e., allocating entries almost sequentially).

Deduplication Metadata Entry. In the LMT, each entry consists of 8 bytes *blocknr* (block number), 8 bytes *fp* (fingerprint), 8 bytes *refcnt* (reference count), and 8 bytes *hint* (used for speculative prefetch). The *blocknr* field refers to the corresponding block number in the file system, while the *refcnt* field refers to the number of references on a data block. The *fp* field stores the 8 bytes xxHash as the block's fingerprint. And the *hint* field contains 61 bits for the location of the subsequent speculated block's entry and 3 bits for *trust degree*.

Region-based Layout. To maintain the locality of deduplication metadata, we group metadata entries into *regions* and allocate entries in the currently used *region* almost sequentially. We use an in-DRAM variable ***Cur Region*** to represent this in-NVM region. For brevity, we do not distinguish *Cur Region* from the currently used in-NVM region. Each *region* is 4 KiB (aligned to the block size) so that the *regions* can be allocated by NVM file systems' block allocator directly [17,66]. The *regions* are linked by 8 bytes pointers at the end of each *region*, and the first a few *regions* (*Region Header*) are reserved in a fixed place in NVM as the list head. In this way, the deduplication metadata table can grow dynamically. Therefore, Light-Dedup avoids unnecessary storage consumption and can scale flexibly when the storage size changes.

We regard a *region* as allocatable (i.e., we can use the *region* to allocate entries) if no more than half of the metadata entries (i.e., $4\,\text{KiB}/32\,\text{Byte}/2 = 64$ entries) are used in that region. Such design is a tradeoff between maintaining the locality of metadata entries' allocation and the space utilization of the region. To allocate regions in constant time, Light-Dedup maintains the positions of allocatable regions with the *Region Queue* in DRAM and keeps track of the number of valid entries in each region with an XArray [60]. Figure 6 illustrates the allocation and deletion of metadata entries:

- **Entry Allocation**. Light-Dedup checks the entries in the *Cur Region* one by one (①) until a free one is found (②), and then returns its position to the caller (③). If there is no free entry found (④), Light-Dedup evicts the *Cur Region* (⑤) and checks if the *Region Queue* is empty. If not, Light-Dedup takes out an allocatable region from the *Region Queue* as the new *Cur Region* (⑥(1)). Otherwise, Light-Dedup leverages the file systems' block allocator to allocate a new *region* as the *Cur Region*, and then links it into the tail of the in-NVM *region* list (⑥(2)). For the new *Cur Region*, Light-Dedup checks the entries in it one by one until a free entry is found (⑦) and returns its position to the caller (⑧).
- **Entry Deletion**. Light-Dedup first sets the *blocknr* of the target entry (⑨) to zero. If exactly half of the metadata entries in that *region* are free, then we insert this *region* back to the *Region Queue* to make it allocatable again (⑩). For the simplicity of concurrent control, the *region* will not be returned to the block allocator.

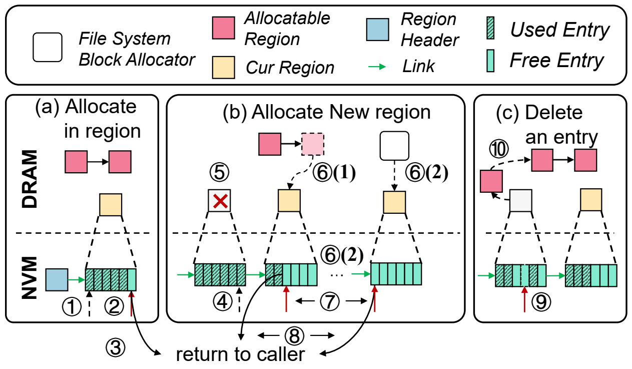


Figure 6: Illustration of region-based entry management. The cross-mark indicates that the *Cur Region* is evicted.

Reduction of Metadata I/O Amplification. To show this, we make the following analysis: (1) **For unique writes**, Light-Dedup allocates a metadata entry for each write almost sequentially in the *Cur Region*, so that the writes to these entries usually hit the NVM write buffer, and thus the metadata write amplification is reduced. (2) **For duplicate writes**, we assume the redundant data tend to be clustered, such as using *cp* or *rsync* to copy a file multiple times. In that case, since the file's deduplication metadata is allocated almost sequentially in a region, the subsequent accesses/modifications to them are also almost sequential, and thus the metadata read and write amplification are both reduced. The extensive study in §5.5 validates our analysis.

Avoidance of Garbage Collection (GC). Maintaining sequentiality (e.g., log-structured layout) often requires GC [66]. However, GC is complex and time-consuming. To address the issue, Light-Dedup does not reclaim allocated regions and allows to reuse them when half of the entries are free. In other words, Light-Dedup trades more NVM space for GC-free design. Such design does not bring significant storage consumption: assuming the capacity of NVM is x, then there are at most $\frac{x}{4\text{KiB}}$ unique blocks to be referenced by Light-Dedup.

Thus, at most $\frac{2x}{4\,\text{KiB}}$ entries are needed for GC-free design (i.e., all the allocated regions are half-full). Since each entry is 32 bytes, at most $\frac{2x}{4\,\text{KiB}} \times 32\,\text{Byte}/x \approx 1.56\%$ space of NVM is needed. Note that this is the worst case. The experiment shows that writing a 128 GiB non-duplicate file allocates 1.008 GiB of regions, which is only 0.79% of the data size.

4.5 Crash Consistency and Recovery

Light-Dedup maintains crash consistency lazily by collaborating with NVM file systems' recovery process. The lazy strategy guarantees crash consistency and avoids eager consistency overheads [9, 47, 68].

Normal Recovery: *Storing in-DRAM index, allocator, etc. to NVM and Reloading them back*. During the clean unmount, Light-Dedup stores in-DRAM *rhashtable* items and the valid entry counts in the reserved area in NVM. During the subsequent remount, Light-Dedup first initializes an empty index, and then inserts the saved items into it. Next, the valid entry counts are loaded into DRAM directly and the *Region Queue* is rebuilt accordingly. After this process, Light-Dedup is ready to accept new I/O requests.

Failure Recovery: *Fixing inconsistency of deduplication metadata in NVM and Reconstructing in-DRAM data structures*. To recover to the normal state, Light-Dedup scans the deduplication metadata during the file systems' recovery to fix two inconsistent cases: (1) A block is only referenced by the file system. It is a non-dedup block, so Light-Dedup does not reinsert it into the deduplication metadata table. (2) A block is only referenced by the deduplication metadata table. Since Light-Dedup treats the file system's metadata as the true source of information, it invalidates the corresponding deduplication metadata entry. After this, Light-Dedup rebuilds its in-DRAM structures similar to the normal recovery.

4.6 Portability

Port to Future NVM Devices. Although Optane DCPMM exited the market recently, we are still confident about NVM technology because it bridges the performance gap between DRAM and SSDs. Our work focuses on the common features of storage-type NVMs as discussed in §2.1, e.g., long media read latency, memory interface, and coarse media access granularity. Therefore, we believe our work can be applied to future commercial storage-type NVMs.

Port to Future CXL-based Devices. Compute Express Link (CXL) [15, 25, 53] is an emerging interconnect standard. We believe that Light-Dedup can also be applied to the storage systems using CXL (e.g., NVMs interconnected with CXL) if the systems exhibit the common features that Light-Dedup focuses on. We leave this as our future work.

Port to Other Instruction Sets. Although Light-Dedup is currently implemented on x86, the idea of hiding long NVM media read latency with speculative prefetch can be applied to other instruction sets with prefetch instructions, such as ARM with the PRFM instruction [2].

5 Performance Evaluation

This section seeks to answer the following questions: (*i*) How does Light-Dedup perform against state-of-the-art NVM (deduplication) file systems? (§5.2) (*ii*) How does Light-Dedup perform in real-world scenarios? (§5.3) (*iii*) How does speculative prefetch in LRBI contribute to final performance? (§5.4) (*iv*) How efficient is the design of LMT? (§5.5) (*v*) How expensive is the Light-Dedup recovery mechanism? (§5.6)

5.1 Experimental Setup

Testbed. We evaluate Light-Dedup on a server with an Intel Xeon Gold 5218 CPU clocked at 2.3 GHz, which has 16 cores (32 hyper-threads) and 22 MiB of L3 cache with *clwb* support. The machine is equipped with 512 GiB Optane DCPMM (2×256 GiB DIMMs) in non-interleaved AppDirect Mode, and 128 GiB DRAM (4 × 32 GiB DIMMs). The server runs CentOS with kernel 5.1.0 modified by NOVA [48].

Compared Systems. We compare Light-Dedup with NOVA, NV-Dedup, DeNOVA, and LD-w/o-P. Among them, the source code of NV-Dedup is not publicly available, thus we re-implement it on top of NOVA[3], following the same configurations in their paper [58]. For DeNOVA, we implement the Deduplication Daemon (DD) based on the open-source version and deduplicate the data in the background aggressively (i.e., *DeNOVA-Immediate* [28]).

Methodology. FIO [4] is used to measure extensive I/O performance. We use *sync* as I/O engine to guarantee the persistence; 0–75% duplication ratio is emulated with parameter *dedupe_percentage*. For the 100% duplication ratio, we perform the same FIO twice and measure the performance of the second run. The reason is that 100% *dedupe_percentage* results in issuing a few unique blocks, which is quite different from the real scenarios. Moreover, we also measure four real-world workloads: copying compiled Linux kernel as code archiving scenario, replaying three realistic traces as frequent data operations scenarios. Among these traces, *WebVMs* and *Mails* are collected from FIU [27, 33]. *Homes* is generated from 50 students' home directories on our OS Lab server: we break the files into 4 KiB blocks, generate each of them md5 digest similar to FIU traces, and use the files' creation time as timestamps. Each measurement is repeated 5 times, and the average values are presented. All coefficients of variation are less than 5%, which suggests reproducibility and stability. The evaluation scripts are available at https://github.com/Light-Dedup/tests.

5.2 Microbenchmarks

We use FIO to evaluate the write performance of Light-Dedup under the different duplication ratios, write patterns, and concurrency levels, i.e., a single thread and 8 threads. Adding more threads does not contribute to the performance improvement due to the contention on the Optane DCPMM [67]. The workload data size is set to 128 GiB to observe a more stable result. Note that under the 100% duplication ratio, each

[3]https://github.com/Light-Dedup/nv-dedup

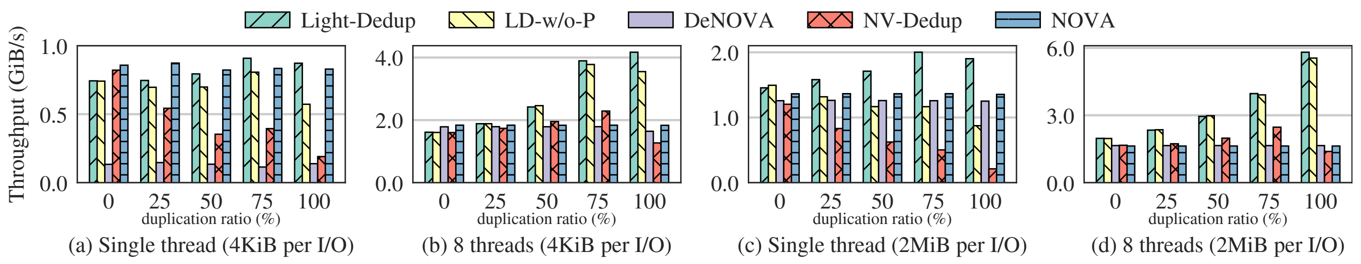


Figure 7: Microbenchmark with FIO under different write patterns (4 KiB/2 MiB per I/O) and concurrency levels.

run performs 64 GiB writes. Further, to better support the large workload in FIO, we modify NOVA by removing the threshold of log extending and by making the extension always double the size of the inode log, which eliminates a substantial amount of many unnecessary *Fast GC*s [66]. The experimental results are shown in Figures 7.

Block-based I/O (4 KiB). Figures 7(a) and (b) represent the throughput of 4 KiB I/O writes. We find that: (1) Light-Dedup achieves 1.70–4.58× throughput over NV-Dedup when the duplication ratio is ≥75% since NV-Dedup suffers from cryptographic hash calculation overheads, while LRBI enables the efficient deduplication in either concurrency levels. (2) Light-Dedup is 3–15% slower than NOVA for the 0% duplication ratio, but the throughput can be 1.05–2.28× to the throughput of NOVA when the duplication ratio is ≥75%. (3) *IBP* contributes to 1–52% performance improvement under single thread compared to LD-w/o-P. However, its contribution is reduced with the increasing threads number due to dramatically enlarged read/write asymmetry; thus, the improvement of *IBP* is diluted. Notably, *CBP* cannot work ideally across syscalls. A possible reason is that the load queue [49] is flushed during the context switch.

Continuous Block I/O (2 MiB). Figures 7(c) and (d) represent the throughput of 2 MiB I/O writes, i.e., writing 512 4 KiB blocks in one syscall. We find that: (1) all the evaluated file systems gain higher write performance due to fewer syscalls. Among them, DeNOVA benefits the most since the contention of DD's dequeue and enqueue decreases significantly. (2) Under the single thread, Light-Dedup achieves 72–118% performance improvement when the duplication ratio is ≥75% compared to LD-w/o-P since *Cross-Block Prefetch* can leverage the locality of workloads to speculate the subsequent block efficiently in a single syscall.

Read-Write Interference. Although Light-Dedup trades slow duplicate writes for faster asynchronous reads, the mixed read/write I/O under multi-thread environments can potentially interfere with each other. This is because writing duplicate blocks require content comparisons and do not write redundant data, which can be seen as a reader-like operation. Conversely, threads that write unique blocks can be seen as writer-like because they aim to write new blocks. Such interference has been previously observed in NyxCache [61] and MT [69]. However, the goal of Light-Dedup is to improve overall performance by eliminating duplicate data blocks, and

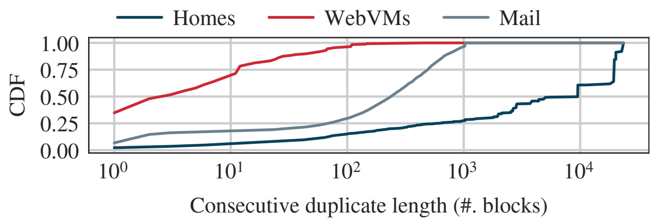


Figure 8: Cumulative distribution function of duplication continuousness of evaluated traces.

Table 3: Characteristics of evaluated real-world workloads.

Workload	Total I/O	Write Prop.	Dup Ratio	Granularity
Copy	13.85 GiB	100%	100%	2 MiB
Homes	63.52 GiB	100%	84%	4 KiB for Blk Max 2 MiB for Bat
WebVMs	54.53 GiB	78%	47%	4 KiB for Blk Max 2 MiB for Bat
Mails	173.27 GiB	91%	95%	4 KiB for Blk Max 2 MiB for Bat

we argue that even though individual tasks can be negatively affected, the overall performance can still be improved.

To show this, we run a set of experiments (not shown in the figure) on the case of 50% duplication ratio and 8 threads. To obtain the separate bandwidth of readers and writers, we make four threads write the duplicated data (as readers) while the remaining four write unique data (as writers). We observe that the bandwidth of Light-Dedup decreases to 1316 MiB/s for readers and 1052 MiB/s for writers when compared to the bandwidth of non-interfered systems with 4 readers (3380 MiB/s) and 4 writers (1608 MiB/s), respectively. However, when compared to the bandwidth of non-interfered systems with 8 writers (1624 MiB/s), the overall bandwidth of Light-Dedup with mixed 4 readers and 4 writers is improved to 2368 MiB/s. Experimental results show that Light-Dedup's non-cryptographic-hash-based deduplication approach can improve overall deduplication performance even in the presence of read-write interference.

5.3 Real-world Scenarios

In this subsection, we study the performance of Light-Dedup under real-world scenarios. The characteristics of the four workloads are summarized in Table 3. Specifically, **for *Copy***, we copy compiled Linux kernel twice from SSD to NVM, and the bandwidth of the second copying is measured. *Copy* can be considered as a real-world application that uses NVM as the storage of code repositories. **For the other three traces**,

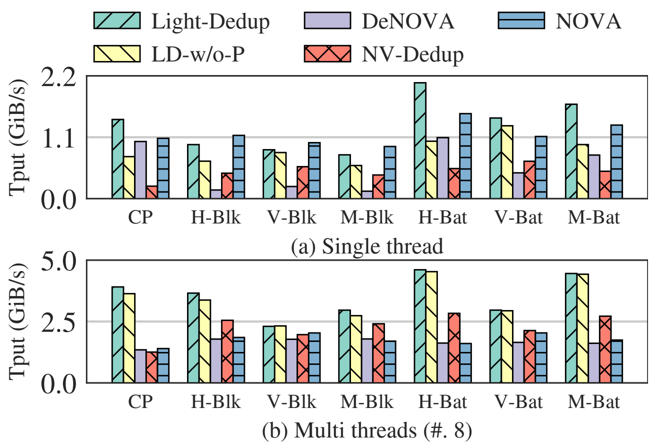


Figure 9: Performance comparison of real-world scenarios. Here, H, V, M indicate *Homes*, *WebVMs*, and *Mails*, respectively. Suffix "-Blk" (***block replay***) and "-Bat" (***batch replay***) indicate that each I/O processes one block and a batch of no more than 512 consecutive blocks, respectively.

we implement *trace-replayer*[4] to emulate the behaviors of a real-world application and replay traces. Unlike previous block-level trace replay tools [27, 33], *trace-replayer* can batch consecutive logical blocks in one syscall, which has the same rationale as many real-world applications (e.g., batching the reads/writes using a local buffer). The insights can be reflected in Figure 8. *Homes* and *Mails* traces show good spatial locality, which the speculative prefetch can well exploit. For *WebVMs*, there are only 30% duplicate blocks whose consecutive length is >10 blocks (point $(10, 0.70)$), which shows relatively poor continuousness.

Figure 9 compares the throughput of Light-Dedup and the other four approaches under different real-world workloads. For DeNOVA, we configure *trace-replayer* to replay the traces by appending data but ignoring the given data offset due to its bugs of handling overlapping writes. Light-Dedup shows the best deduplication performance under all workloads in most cases. In particular, (1) during *Copy*, Light-Dedup is much faster than other approaches since speculative prefetch markedly hides read latency under a single thread (due to the continuousness of the workload). (2) During *block reply*, *In-Block Prefetch* shows its effectiveness under a single thread. With the increasing number of threads and the enlarged read/write asymmetry, the throughput of LD-w/o-P catches up with that of Light-Dedup. (3) During *batch replay*, speculative prefetch contributes a lot to single-thread performance with a high duplication ratio. For example, for single-thread *Mails*, Light-Dedup achieves 1.28× write throughput compared to NOVA under a single thread. However, LD-w/o-P cannot even catch up with NOVA's throughput.

5.4 Speculative Prefetch Efficiency

In this subsection, we study the efficiency of prefetching and speculation by using FIO with block size set to 2 MiB, and *Copy* as our benchmarks.

Single-thread Comparison. Figure 10(a) writes the same 64 GiB data twice (using **FIO**) with a single thread, and the second writing bandwidth is measured. *PN* significantly exceeds other variants by 1.11–2.19×, mainly because prefetch enables the parallelism of CPU calculation and NVM I/O and significantly reduces content-comparison time. We further measure the deduplication performance of *Copy* under a single thread. Figure 10(b) presents the throughput of the second copy. The result is similar to FIO, except the overall throughput is lower. This is because there are 44% small files (less than 4 KiB) in the Linux kernel source code, and these small files degrade the deduplication performance.

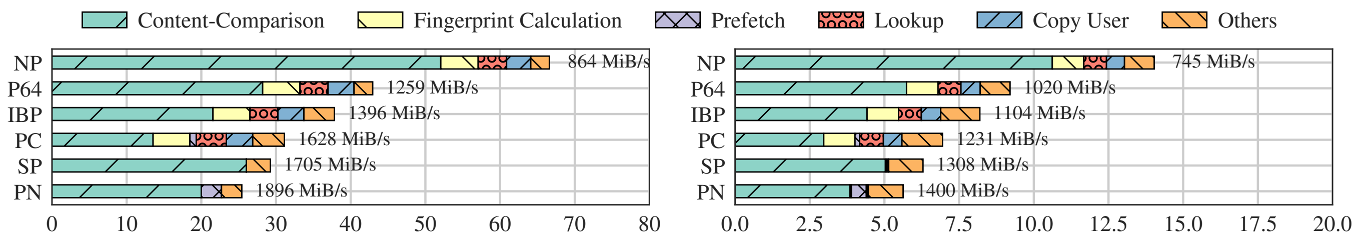
Multi-thread Comparison and Observations. To study how *PN* scales with the increasing concurrency level, we use the same FIO benchmark but with the number of threads set to 1–16. Figure 11 presents the second write performance. The figure shows that: (1) The throughput of *PN* is 1.03–1.29× and 1.51–2.19× that of *SP* and LD-w/o-P when the threads number ≤ 5 since *PN* prefetches NVM data into the CPU cache, which reduces content-comparison time. (2) When the threads number ≥ 6, *SP* shows about 1.11–1.80× throughput compared to *PN*. According to the breakdown performance (not shown in the figure due to space limits), we find that the content-comparison time of *PN* rises up to 1.65× to that of *SP* under 16 threads, which suggests the large amount of extra prefetch I/O exacerbates the contention of in-NVM buffers and thus leads to longer I/O latency. (3) The evaluation shows that *CBP* (i.e., *PN*+*Transition*, see §4.3) combines the benefits of *SP* and *PN* and scales well with the increment of threads.

5.5 Metadata I/O Amplification in LMT

To study the efficiency of region-based layout in LMT, we have designed a workload to quickly age the file system. (1) We first write a 128 GiB file (2 MiB per I/O) to a newly mounted deduplication file system (Fresh System). (2) Next, we punch the file randomly by using `fallocate()` until the file size is reduced to half. This step creates random holes in the file system, which emulates the aging process (Aged System). (3) Finally, we write another 64 GiB file to fill the holes. In the aged system, the spatial distribution of free entries is random. Inappropriate metadata management (e.g., *entry-based* layout used in NV-Dedup [58]) can cause severe read/write amplification and consequently decline the system's I/O performance.

Table 4 shows the comparison between *region-based* and *entry-based* metadata layout in multiple dimensions under the aging workload. The evaluation shows that *region-based* consistently outperforms *entry-based* in all the dimensions and can resist fragmentation problems. Maintaining the locality of entries significantly reduces the write amplification under the aged system (i.e., from *entry-based*'s 9.86× to *region-based*'s 3.43×), and improves about 11.6% write throughput relative to *entry-based*. The results suggest that *region-based* meta-

[4]The tool is available at https://github.com/Light-Dedup/nvm_tools



(a) FIO Breakdown (in seconds) (b) Copy compiled Linux Breakdown (in seconds)

Figure 10: Performance comparison and breakdowns of different variants. For simplicity, we denote LD-w/o-P as *NP* (i.e., no speculative prefetch), and recall that IBP is *In-Block Prefetch*. Note that *Cross-Block Prefetch (CBP)* is effectively equivalent to *PN* in single-thread experiments; thus, it is omitted in the above figures.

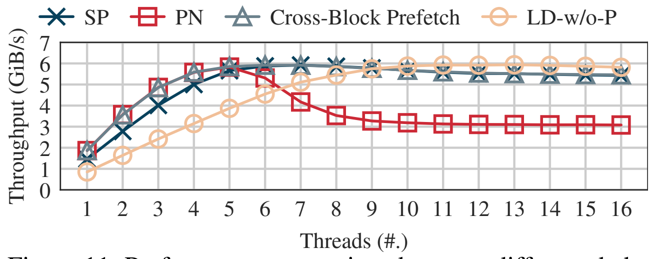


Figure 11: Performance comparison between different deduplication strategies under different threads.

Table 4: Comparison of region-based and entry-based metadata layout under the aging workload.

Dimension	Fresh System (128 GiB)		Aged System (64 GiB)	
	Region	Entry	**Region**	Entry
Reads Per Block (B)	116.28 (2.91×)	126.94 (3.17×)	**244.19 (6.1×)**	774.13 (19.35×)
Writes Per Block (B)	75.75 (1.89×)	79.56 (1.99×)	**137.17 (3.43×)**	394.54 (9.86×)
Throughput (MiB/s)	1747.5	1690.6	**1336.72**	1197.76

data layout is more friendly to NVM deduplication, especially in an aged file system (which is common in the production environments).

5.6 Recovery Overheads

Table 5 studies the unmount time, normal recovery time, and failure recovery time of NOVA and Light-Dedup with different sizes of files (the total number of files is fixed to 32). The results show that although Light-Dedup causes overheads during recovery, its unmount and failure recovery time remains the same trend as NOVA (linearly) as the file size grows. We argue that trading longer unmount/recovery time for more efficient runtime is reasonable for NVM deduplication.

6 Discussion

Memory Consumption of *rhashtable*. We observe that the memory consumption of writing 128 GiB data under 0%, 25%, 50%, and 75% duplication ratio is 1.26 GiB, 1.08 GiB, 658 MiB, and 331 MiB, respectively. The experimental results indicate that the memory consumption is less than 1% of the data size (e.g., $1.26GiB/128GiB \approx 0.98\%$).

Hardware-based Cryptographic Hash Calculation. There are several hardware accelerators developed for efficiently calculating cryptographic hash [5, 14, 42, 55]. However, they are not widely deployed and require special hardware. Therefore, they are not considered in this paper.

Table 5: Comparison of recovery overheads.

Dimension	File system	File system utilization (GiB)		
		32 × 1	32 × 2	32 × 4
Umount Time (s)	NOVA	0.385	0.775	1.502
	Light-Dedup	0.551	1.095	2.099
Normal Recovery Time (s)	NOVA	0.015	0.015	0.015
	Light-Dedup	0.617	1.223	2.398
Failure Recovery Time (s)	NOVA	0.315	0.488	0.829
	Light-Dedup	1.260	2.372	4.604

Scalability on Multiple Optane DCPMMs. We run a 32 GiB FIO workload with 75% duplication ratio on two interleaved 256 GiB DCPMMs. The experimental results show that the throughput of Light-Dedup increases from 952 MiB/s to 6238 MiB/s with 1–16 threads, suggesting Light-Dedup can scale with increasing threads on multiple Optane DCPMMs.

7 Conclusion and Future Work

In this paper, we propose Light-Dedup, a light-weight inline deduplication framework for NVM file systems. With the NVM-aware Light-Redundant-Block-Identifier (LRBI) and Light-Meta-Table (LMT), Light-Dedup is able to maximize NVM deduplication performance by fully considering NVM's I/O mechanisms (e.g., long media read latency). Evaluation results show that the deduplication cost is low, and the performance can be enhanced if the duplication ratio is high. Since memory usage is sensitive to server environment [28], we plan to incorporate other memory-efficient hash table design [36, 45, 73, 74] to optimize Light-Dedup's index further.

Acknowledgments

We thank our shepherd, Youjip Won, and the anonymous reviewers for their constructive comments and insightful suggestions. This research was partly supported by the National Natural Science Foundation of China under Grant no. 61972441 and U22B2022; Shenzhen Science and Technology Innovation Program under Grant no. RCYX20210609104510007; Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515012634.

References

[1] Mijin An, Soojun Im, Dawoon Jung, and Sang-Won Lee. Your read is our priority in flash storage. *Proceedings of the VLDB Endowment*, 15(9):1911–1923, 2022.

[2] ARM. Arm cortex-a75 core technical reference manual r2p0, 2023. https://developer.arm.com/documentation/100403/0200/functional-description/level-1-memory-system/data-prefetching.

[3] Amro Awad, Sergey Blagodurov, and Yan Solihin. Write-aware management of NVM-based memory extensions. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2925426.2926284.

[4] Jens Axboe. Flexible i/o tester, 2017. https://github.com/axboe/fio.

[5] Sergei Brazhnikov. A hardware implementation of the SHA2 hash algorithms using CMOS 28nm technology. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 1784–1786. IEEE, 2020.

[6] Mu-Tien Chang, Paul Rosenfeld, Shih-Lien Lu, and Bruce Jacob. Technology comparison for large last-level caches (l 3 cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM. In *2013 IEEE 19th international symposium on high performance computer architecture (HPCA)*, pages 143–154. IEEE, 2013.

[7] Wande Chen, Zhenke Chen, Dingding Li, Hai Liu, and Yong Tang. Low-overhead inline deduplication for persistent memory. *Transactions on Emerging Telecommunications Technologies*, page e4079, 2020.

[8] Youmin Chen, Youyou Lu, Bohong Zhu, et al. Scalable persistent memory file system with kernel-userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 81–95, 2021.

[9] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Consistency without ordering. In *FAST*, page 9, 2012.

[10] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1683–1694, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Yann Collet. xxhash: Extremely fast hash algorithm, 2016. https://github.com/Cyan4973/xxHash.

[12] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[13] Jonathan Corbet. Relativistic hash tables, part 1: Algorithms, 2014. https://lwn.net/Articles/612021/.

[14] Luigi Dadda, Marco Macchetti, and Jeff Owen. The design of a high speed asic unit for the hash function sha-256 (384, 512). In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 3, pages 70–75. IEEE, 2004.

[15] Debendra Das Sharma. Keynote 1: Compute express link (cxl) changing the game for cloud computing. In *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages xii–xii, 2021.

[16] Biplob K Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX annual technical conference (USENIX ATC'10)*, pages 1–16, 2010.

[17] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*, pages 1–15, 2014.

[18] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):855–868, 2016.

[19] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 181–192, Philadelphia, PA, June 2014. USENIX Association.

[20] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 331–344, Santa Clara, CA, February 2015. USENIX Association.

[21] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. Energy-efficient hybrid DRAM/NVM main memory. In *Proceedsings of 2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 492–493, 2015.

[22] Intel. ipmctl, 2018. https://github.com/intel/ipmctl.

[23] Intel. Intel® 64 and ia-32 architectures software developer manuals, 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[24] Joseph Izraelevitz, Jian Yang, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[25] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 45–51, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Myoungsoo Jung, John Shalf, and Mahmut Kandemir. Design of a large-scale storage-class RRAM system. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 103–114, New York, NY, USA, 2013. Association for Computing Machinery.

[27] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[28] Hyungjoon Kwon, Yonghyeon Cho, et al. Denova: Deduplication extended nova file system. In *IPDPS*, pages 1–12. IEEE, 2022.

[29] Giusy Lama. *Phase Change Memory (PCM) for High Density Storage Class Memory (SCM) Applications*. PhD thesis, Université Grenoble Alpes [2020-], 2022.

[30] Hyokeun Lee, Moonsoo Kim, Hyunchul Kim, Hyun Kim, and Hyuk-Jae Lee. Integration and boost of a read-modify-write module in phase change memory system. *IEEE Transactions on Computers*, 68(12):1772–1784, 2019.

[31] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. In *Asian Symposium on Programming Languages and Systems*, pages 244–265. Springer, 2019.

[32] Jingwei Li, Zuoru Yang, et al. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, pages 1–15, 2020.

[33] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, February 2016. USENIX Association.

[34] Yu-Pei Liang, Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Pei-Yu Chen, and Wei-Kuan Shih. Rethinking last-level-cache write-back strategy for mlc stt-ram main memory with asymmetric write energy. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019.

[35] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[36] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable hashing on persistent memory. In *Proceedings of the VLDB Endowment*, page 1147–1161, April 2020.

[37] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. Pod: Performance oriented i/o deduplication for primary storage systems in the cloud. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 767–776. IEEE, 2014.

[38] Paul McKenney. What is rcu, fundamentally?, 2007. https://lwn.net/Articles/262464/.

[39] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

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

[41] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *Supercomputing frontiers and innovations*, 1(3):19–55, 2014.

[42] Rahul P Naik and Nicolas T Courtois. Optimising the SHA256 hashing algorithm for faster and more efficient bitcoin mining. *MSc Information Security Department of Computer Science UCL*, pages 1–65, 2013.

[43] Prashant J Nair, Chiachen Chou, Bipin Rajendran, and Moinuddin K Qureshi. Reducing read latency of phase change memory via early read and turbo read. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 309–319. IEEE, 2015.

[44] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. *ACM SIGPLAN Notices*, 52(4):135–148, 2017.

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

[46] Heba Nashaat, Nesma Ashry, and Rawya Rizk. Smart elastic scheduling algorithm for virtual machine migration in cloud computing. *The Journal of Supercomputing*, 75(7):3842–3865, 2019.

[47] Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui. Live deduplication storage of virtual machine images in an open-source cloud. In *Proceedings of Middleware 2011*, pages 81–100, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[48] NVSL. Nova: Non-volatile memory accelerated log-structured file system, 2017. https://github.com/NVSL/linux-nova.

[49] Il Park, Chong Liang Ooi, and TN Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 411–422. IEEE, 2003.

[50] Moinuddin K Qureshi, Michele M Franceschini, and Luis A Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–11. IEEE, 2010.

[51] Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-Montaño. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–11, 2010.

[52] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020.

[53] Debendra Das Sharma. Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 5–12, 2022.

[54] ChunYi Su, David Roberts, Edgar A León, Kirk W Cameron, Bronis R de Supinski, Gabriel H Loh, and Dimitrios S Nikolopoulos. Hpmc: An energy-aware management system of multi-level memory architectures. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 167–178, 2015.

[55] Vikram Suresh, Sudhir Satpathy, Sanu Mathew, Mark Anders, Himanshu Kaul, Amit Agarwal, Steven Hsu, and Ram Krishnamurthy. A 230mv-950mv 2.8 tbps/w unified sha256/sm3 secure hashing hardware accelerator in 14nm tri-gate cmos. In *ESSCIRC 2018-IEEE 44th European Solid State Circuits Conference (ESSCIRC)*, pages 98–101. IEEE, 2018.

[56] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *Proceedings of the 2014 Ottawa Linux Symposium (OLS'14)*, pages 83–95. Citeseer, 2014.

[57] K. Venkatesh and D. Narasimhan. Revealing the novel precise subset identification and deduplication of audio substance over the shared public environment. *The Journal of Supercomputing*, Feb 2022.

[58] Chundong Wang, Qingsong Wei, Jun Yang, Cheng Chen, Yechao Yang, and Mingdi Xue. Nv-dedup: High-performance inline deduplication for non-volatile memory. *IEEE Transactions on Computers*, 67(5):658–671, 2017.

[59] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[60] Matthew Wilcox. Xarray. https://www.kernel.org/doc/html/v5.1/core-api/xarray.html.

[61] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, Santa Clara, CA, February 2022. USENIX Association.

[62] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[63] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX annual technical conference (USENIX ATC'11)*, pages 26–30, 2011.

[64] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Cong Xu, Xiangyu Dong, Norman P Jouppi, and Yuan Xie. Design implications of memristor-based RRAM cross-point structures. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[66] Jian Xu and Steven 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'16)*, pages 323–338, 2016.

[67] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 169–182, 2020.

[68] Qirui Yang, Runyu Jin, and Ming Zhao. Smartdedup: optimizing deduplication for resource-constrained devices. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 633–646, 2019.

[69] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT^2: Memory bandwidth regulation on hybrid nvm/dram platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, 2022.

[70] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. Htmfs: Strong consistency comes for free with hardware transactional memory in persistent memory file systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 17–34, 2022.

[71] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting PCM asymmetries. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 282–293, 2013.

[72] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 269–282, 2008.

[73] Xiaomin Zou, Fang Wang, Dan Feng, Chaojie Liu, Fan Li, and Nan Su. Hmeh: write-optimal extendible hashing for hybrid DRAM-NVM memory. In *2020 36th Symposium on Mass Storage Systems and Technologies (MSST)*, 2020.

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

[75] Pengfei Zuo, Yu Hua, Ming Zhao, Wen Zhou, and Yuncheng Guo. Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2018)*, pages 442–454. IEEE, 2018.