iDedup: Latency-aware, inline data deduplication for primary storage

Kiran Srinivasan, Tim Bisson, Garth Goodson, Kaladhar Voruganti

NetApp, Inc.
{skiran, tbisson, goodson, kaladhar}@netapp.com

Abstract

Deduplication technologies are increasingly being deployed to reduce cost and increase space-efficiency in corporate data centers. However, prior research has not applied deduplication techniques inline to the request path for latency sensitive, primary workloads. This is primarily due to the extra latency these techniques introduce. Inherently, deduplicating data on disk causes fragmentation that increases seeks for subsequent sequential reads of the same data, thus, increasing latency. In addition, deduplicating data requires extra disk IOs to access on-disk deduplication metadata. In this paper, we propose an inline deduplication solution, iDedup, for primary workloads, while minimizing extra IOs and seeks.

Our algorithm is based on two key insights from real-world workloads: i) spatial locality exists in duplicated primary data; and ii) temporal locality exists in the access patterns of duplicated data. Using the first insight, we selectively deduplicate only sequences of disk blocks. This reduces fragmentation and amortizes the seeks caused by deduplication. The second insight allows us to replace the expensive, on-disk, deduplication metadata with a smaller, in-memory cache. These techniques enable us to tradeoff capacity savings for performance, as demonstrated in our evaluation with real-world workloads. Our evaluation shows that iDedup achieves 60-70% of the maximum deduplication with less than a 5% CPU overhead and a 2-4% latency impact.

1 Introduction

Storage continues to grow at an explosive rate of over 52% per year [10]. In 2011, the amount of data will surpass 1.8 zettabytes [17]. According to the IDC [10], to reduce costs and increase storage efficiency, more than 80% of corporations are exploring deduplication technologies. However, there is a huge gap in the current capabilities of deduplication technology. No deduplication systems exist that deduplicate *inline* with client requests for latency sensitive primary workloads. All prior deduplication work focuses on either: i) throughput sensitive archival and backup systems [8, 9, 15, 21, 26, 39, 41]; or ii) latency sensitive primary systems that deduplicate data *offline* during idle time [1, 11, 16]; or iii) file systems with inline deduplication, but agnostic to performance [3, 36]. This paper introduces two novel insights that enable latency-aware, *inline*, primary deduplication.

Many primary storage workloads (e.g., email, user directories, databases) are currently unable to leverage the benefits of deduplication, due to the associated latency costs. Since offline deduplication systems impact latency the least, they are currently the best option; however, they are inefficient. For example, offline systems require additional storage capacity to absorb the writes prior to deduplication, and excess disk bandwidth to perform reads and writes during deduplication. This additional disk bandwidth can impact foreground workloads. Additionally, inline compression techniques also exist [5, 6, 22, 38] that are complementary to our work.

The challenge of inline deduplication is to not increase the latency of the already latency sensitive, foreground operations. Reads are affected by the fragmentation in data layout that naturally occurs when deduplicating blocks across many disks. As a result, subsequent sequential reads of deduplicated data are transformed into random IOs resulting in significant seek penalties. Most of the deduplication work occurs in the write path; i.e., generating block hashes and finding duplicate blocks. To identify duplicates, on-disk data structures are accessed. This leads to extra IOs and increased latency in the write path. To address these performance concerns, it is necessary to minimize any latencies introduced in both the read and write paths.

We started with the realization that in order to improve latency a tradeoff must be made elsewhere. Thus, we were motivated by the question: *Is there a tradeoff between performance and the degree of achievable dedu-*

plication? While examining real-world traces [20], we developed two key insights that ultimately led to an answer: i) spatial locality exists in the duplicated data; and ii) temporal locality exists in the accesses of duplicated data. The first observation allows us to amortize the seeks caused by deduplication by only performing deduplication when a sequence of on-disk blocks are duplicated. The second observation enables us to maintain an in-memory fingerprint cache to detect duplicates in lieu of any on-disk structures. The first observation mitigates fragmentation and addresses the extra read path latency; whereas, the second one removes extra IOs and lowers write path latency. These observations lead to two control parameters: i) the minimum number of sequential duplicate blocks on which to perform deduplication; and ii) the size of the in-memory fingerprint cache. By adjusting these parameters, a tradeoff is made between the capacity savings of deduplication and the performance impact to the foreground workload.

This paper describes the design, implementation and evaluation of our deduplication system (iDedup) built to exploit the spatial and temporal localities of duplicate data in primary workloads. Our evaluation shows that good capacity savings are achievable (between 60%-70% of maximum) with a small impact to latency (2-4% on average). In summary, our key contributions include:

- Insights on spatial and temporal locality of duplicated data in real-world, primary workloads.
- Design of an inline deduplication algorithm that leverages both spatial and temporal locality.
- Implementation of our deduplication algorithm in an enterprise-class, network attached storage system.
- Implementation of efficient data structures to reduce resource overheads and improve cacheability.
- Demonstration of a viable tradeoff between performance and capacity savings via deduplication.
- Evaluation of our algorithm using data from real-world, production, enterprise file system traces.

The remainder of the paper is as follows: Section 2 provides background and motivation of the work; Section 3 describes the design of our deduplication system; Section 4 describes the system's implementation; Section 5 evaluates the implementation; Section 6 describes related work, and Section 7 concludes.

2 Background and motivation

Thus far, the majority of deduplication research has targeted improving deduplication within the backup and archival (or secondary storage) realm. As shown in Table 1, very few systems provide deduplication for latency sensitive primary workloads. We believe that this is due to the significant challenges in performing deduplication without affecting latency, rather than the lack of benefit deduplication provides for primary workloads. Our system is specifically targeted at this gap.

Type	**Offline**	**Inline**
Primary, latency sensitive	NetApp ASIS [1], EMC Celerra [11], StorageTank [16],	***i*Dedup** (This paper)
Secondary, throughput sensitive	(No motivation for systems in this category)	EMC DDFS [41], EMC Cluster [8] DeepStore [40], NEC HydraStor [9], Venti [31], SiLo [39], Sparse Indexing [21], ChunkStash [7], Foundation [32], Symantec [15], EMC Centera [24], GreenBytes [13]

Table 1: *Table of related work:*. The table shows how this paper, *iDedup*, is positioned relative to some other relevant work. Some primary, inline deduplication file systems (like ZFS [3]) are omitted, since they are not optimized for latency.

The remainder of this section further describes the differences between primary and secondary deduplication systems and describes the unique challenges faced by primary deduplication systems.

2.1 Classifying deduplication systems

Although many classifications for deduplication systems exist, they are usually based on internal implementation details, such as the fingerprinting (hashing) scheme or whether fixed sized or variable sized blocks are used. Although important, these schemes are usually orthogonal to the types of workloads their system supports. Similar to other storage systems, deduplication systems can be broadly classified as *primary* or *secondary* depending on the workloads they serve. Primary systems are used for primary workloads. These workloads tend to be latency sensitive and use RPC based protocols, such as NFS [30], CIFS [37] or iSCSI [35]. On the other hand, secondary systems are used for archival or backup purposes. These workloads process large amounts of data, are throughput sensitive and are based on streaming protocols.

Primary and secondary deduplication systems can be further subdivided into *inline* and *offline* deduplication systems. Inline systems deduplicate requests in the write path before the data is written to disk. Since inline deduplication introduces work into the critical write path, it often leads to an increase in request latency. On the other hand, offline systems tend to wait for system idle time to deduplicate previously written data. Since no operations are introduced within the write path; write latency is not affected, but reads remain fragmented.

Content addressable storage (CAS) systems (e.g., [24, 31]) naturally perform inline deduplication, since blocks are typically addressed by their fingerprints. Both archival and CAS systems are sometimes used for primary storage. Likewise, a few file systems that perform inline deduplication (e.g., ZFS [3] and SDFS [36]) are also used for primary storage. However, none of these systems are specifically optimized for latency sensitive workloads while performing inline deduplication. Their design for maximum deduplication introduces extra IOs and does not address fragmentation.

Primary inline deduplication systems have the following advantages over offline systems:

1. Storage provisioning is easier and more efficient: Offline systems require additional space to absorb the writes prior to deduplication processing. This causes a temporary bloat in storage usage leading to inaccurate space accounting and provisioning.
2. No dependence on system idle time: Offline systems use idle time to perform deduplication without impacting foreground requests. This is problematic when the system is busy for long periods of time.
3. Disk-bandwidth utilization is lower: Offline systems use extra disk bandwidth when reading in the staged data to perform deduplication and then again to write out the results. This limits the total bandwidth available to the system.

For good reason, the majority of prior deduplication work has focused on the design of inline, secondary deduplication systems. Backup and archival workloads typically have a large amount of duplicate data, thus the benefit of deduplication is large. For example, reports of 90+% deduplication ratios are not uncommon for backup workloads [41], compared to the 20-30% we observe from our traces of primary workloads. Also, since backup workloads are not latency sensitive, they are tolerant to delays introduced in the request path.

2.2 Challenges of primary deduplication

The almost exclusive focus on maximum deduplication at the expense of performance has left a gap for latency sensitive workloads. Since primary storage is usually the most expensive, any savings obtained in primary systems has high cost advantages. Due to their higher cost ($/GB), deduplication is even more critical for flash based systems; nothing precludes our techniques from working with these systems. In order for primary, inline deduplication to be practical for enterprise systems, a number of challenges must be overcome:

- Write path: The metadata management and IO required to perform deduplication inline with the write request increases write latency.

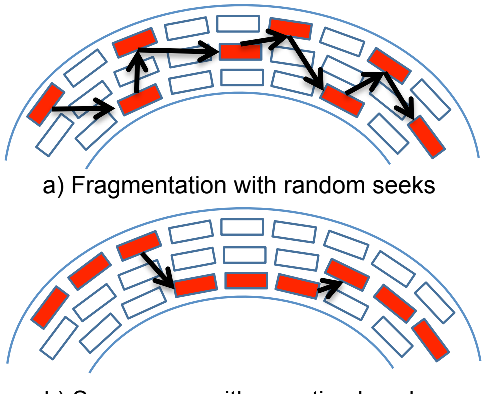

a) Fragmentation with random seeks
b) Sequences, with amortized seeks

Figure 1: *a) Increase in seeks due to increased fragmentation. b) The amortization of seeks using sequences.* This figure shows the amortization of seeks between disk tracks by using sequences of blocks (threshold=3).

- Read path: The fragmentation of otherwise sequential writes increases the number of disk seeks required during reads. This increases read latency.
- Delete path: The requirement to check whether a block can be safely deleted increases delete latency.

All of these penalties, due to deduplication, impact the performance of foreground workloads. Thus, primary deduplication systems only employ offline techniques to avoid interfering with foreground requests [1, 11, 16].

Write path: For inline, primary deduplication, write requests deduplicate data blocks prior to writing those blocks to stable storage. At a minimum, this involves fingerprinting the data block and comparing its signature within a table of previously written blocks. If a match is found, the metadata for the block, e.g., the file's block pointer, is updated to point to the existing block and no write to stable storage is required. Additionally, a reference count on the existing block is incremented. If a match is not found, the block is written to stable storage and the table of existing blocks is updated with the new block's signature and its storage location. The additional work performed during write path deduplication can be summarized as follows:

- Fingerprinting data consumes extra CPU resources.
- Performing fingerprint table lookups and managing the table persistently on disk requires extra IOs.
- Updating a block's reference count requires an update to persistent storage.

As one can see, the management of deduplication metadata, in memory, and on persistent storage, accounts for the majority of write path overheads. Even though much previous work has explored optimizing metadata management for inline, secondary systems (e.g., [2, 15, 21, 39, 41]), we feel that it is necessary to minimize all extra IO in the critical path for latency sensitive workloads.

Read path: Deduplication naturally fragments data that would otherwise be written sequentially. Fragmentation occurs because a newly written block may be deduplicated to an existing block that resides elsewhere on storage. Indeed, the higher the deduplication ratio, the higher the likelihood of fragmentation. Figure 1(a) shows the potential impact of fragmentation on reads in terms of the increased number seeks. When using disk based storage, the extra seeks can cause a substantial increase in read latency. Deduplication can convert sequential reads from the application into random reads from storage.

Delete path: Typically, some metadata records the usage of shared blocks. For example, a table of reference counts can be maintained. This metadata must be queried and updated inline to the deletion request. These actions can increase the latency of delete operations.

3 Design

In this section, we present the rationale that led to our solution, the design of our architecture, and the key design challenges of our deduplication system (iDedup).

3.1 Rationale for solution

To better understand the challenges of inline deduplication, we performed data analysis on real-world enterprise workloads [20]. First, we ran simulations varying the block size to see its effect on deduplication. We observed that the drop in deduplication ratio was less than linear with increasing block size. This implies duplicated data is clustered, thus indicating *spatial locality* in the data. Second, we ran simulations varying the fingerprint table size to determine if the same data is written repeatedly close in time. Again, we observed the drop in deduplication ratio was less than linear with decreasing table size. This implies duplicated data exhibits notable *temporal locality*, thus making the fingerprint table amenable to caching. Unfortunately, we could not test our hypothesis on other workloads due to the lack of traces with data duplication patterns.

3.2 Solution overview

We use the observations of spatial and temporal locality to derive an inline deduplication solution.

Spatial locality: We leverage the spatial locality to perform selective deduplication, thereby mitigating the extra seeks introduced by deduplication for sequentially read files. To accomplish this, we examine blocks at write time and attempt to only deduplicate full sequences of file blocks *if and only if the sequence of blocks are i) sequential in the file and ii) have duplicates that are sequential on disk.* Even with this optimization, sequential reads can still incur seeks between sequences. However, if we enforce an appropriate *minimum sequence length* for such sequences (the *threshold*), the extra seek cost is expected to be amortized; as shown by Figure 1(b). The threshold is a configurable parameter in our system. While some schemes employ a larger block size to leverage spatial locality, they are limited as the block size represents both the minimum and the maximum sequence length. Whereas, our threshold represents the minimum sequence length and the maximum sequence length is only limited by the file's size.

Inherently, due to our selective approach, only a subset of blocks are deduplicated, leading to lower capacity savings. Therefore, our inline deduplication technique exposes a tradeoff between capacity savings and performance, which we observe via experiments to be reasonable for certain latency sensitive workloads. For an optimal tradeoff, the threshold must be derived empirically to match the randomness in the workload. Additionally, to recover the lost savings, our system does not preclude executing other offline techniques.

Temporal locality: In all deduplication systems, there is a structure that maps the fingerprint of a block and its location on disk. We call this the deduplication metadata structure (or dedup-metadata for short). Its size is proportional to the number of blocks and it is typically stored on disk. Other systems use this structure as a lookup table to detect duplicates in the write path; this leads to extra, expensive, latency-inducing, random IOs.

We leverage the temporal locality by maintaining dedup-metadata as a *completely memory-resident, LRU cache*, thereby, avoiding extra dedup-metadata IOs. There are a few downsides to using a smaller, in-memory cache. Since we only cache mappings for a subset of blocks, we might not deduplicate certain blocks due to lack of information. In addition, the memory used by the cache reduces the file system's buffer cache size. This can lead to a lower buffer cache hit rate, affecting latency. On the other hand, the buffer cache becomes more effective by caching deduplicated blocks [19]. These observations expose another tradeoff between performance (hit rate) and capacity savings (dedup-metadata size).

3.3 Architecture

In this subsection, we provide an overview of our architecture. In addition, we describe the changes to the IO path to perform inline deduplication.

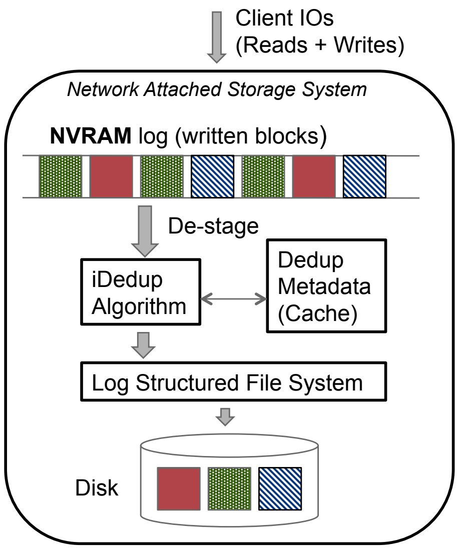

Client IOs
(Reads + Writes)
Network Attached Storage System
NVRAM log (written blocks)
De-stage
iDedup Algorithm
Dedup Metadata (Cache)
Log Structured File System
Disk

Figure 2: *iDedup Architecture.* Non-deduplicated blocks (different patterns) in NVRAM buffer are deduplicated by the iDedup algorithm before writing them to disk via the file system.

3.3.1 Storage system overview

An enterprise-class network-attached storage (NAS) system (as illustrated in Figure 2) is used as the reference system to build iDedup. For primary workloads, the system supports the NFS [30] and CIFS [37] RPC-based protocols. As seen in Figure 2, the system uses a log-structured file system [34] combined with non-volatile RAM (NVRAM) to buffer client writes to reduce response latency. These writes are periodically flushed to disk during the *destage* phase. Allocation of new disk blocks occur during this phase and is performed successively for each file written. Individual disk blocks are identified by their unique *disk block numbers* (DBNs). File metadata, containing the DBNs of its blocks, is stored within an *inode* structure. Given our objective to perform inline deduplication, the newly written (*dirty*) blocks need to be deduplicated during the destage phase. By performing deduplication during destage, the system benefits by not deduplicating short-lived data that is overwritten or deleted while buffered in NVRAM. Adding inline deduplication modifies the write path significantly.

3.3.2 Write path flow

Compared to the normal file system write path, we add an extra layer of deduplication processing. As this layer consumes extra CPU cycles, it can prolong the total time required to allocate dirty blocks and affect time-sensitive file system operations. Moreover, any extra IOs in this layer can interfere with foreground read requests. Thus, this layer must be optimized to minimize overheads. On the other hand, there is an opportunity to overlap deduplication processing with disk write IOs in the destage phase. The following steps take place in the write path:

1. For each file, the list of dirty blocks is obtained.
2. For each dirty block, we compute its fingerprint (hash of the block's content) and perform a lookup in the dedup-metadata structure using the hash as the key.
3. If a duplicate is found, we examine adjacent blocks, using the iDedup algorithm (Section 3.4), to determine if it is part of a duplicate sequence.
4. While examining subsequent blocks, some duplicate sequences might end. In those cases, the length of the sequence is determined, if it is greater than the configured threshold, we mark the sequence for deduplication. Otherwise, we allocate new disk blocks and add the fingerprint metadata for these blocks.
5. When a duplicate sequence is found, the DBN of each block in the sequence is obtained and the file's metadata is updated and eventually written to disk.
6. Finally, to maintain file system integrity in the face of deletes, we update reference counts of the duplicated blocks in a separate structure on disk.

3.3.3 Read path flow

Since iDedup updates the file's metadata as soon as deduplication occurs, the file system cannot distinguish between a duplicated block and a non-duplicated one. This allows file reads to occur in the same manner for all files, regardless of whether they contain deduplicated blocks. Although sequential reads may incur extra seeks due to deduplication, having a minimum sequence length helps amortize this cost. Moreover, if we pick the threshold closer to the expected sequentiality of a workload, then the effects of those seeks can be hidden.

3.3.4 Delete path flow

As mentioned in the write path flow, deduplicated blocks need to be reference counted. During deletion, the reference count of deleted blocks is decremented and only blocks with no references are freed. In addition to updating the reference counts, we also update the in-memory dedup-metadata when a block is deleted.

3.4 iDedup algorithm

The iDedup deduplication algorithm has the following key design objectives:

1. The algorithm should be able to identify sequences of file blocks that are duplicates and whose corresponding DBNs are sequential.
2. The largest duplicate sequence for a given set of file blocks should be identified.
3. The algorithm should minimize searches in the dedup-metadata to reduce CPU overheads.

4. The algorithm execution should overlap with disk IO during the destage phase and not prolong the phase.
5. The memory and CPU overheads caused by the algorithm should not prevent other file system processes from accomplishing their tasks in a timely manner.
6. The dedup-metadata must be optimized for lookups.

More details of the algorithm are presented in Section 4.2. Next, we describe the design elements that enable these objectives.

3.4.1 Dedup-metadata cache design

The dedup-metadata is maintained as a cache with one entry per block. Each entry maps the fingerprint of a block to its DBN on disk. We use LRU as the cache replacement policy; other replacement policies did not perform better than the simpler LRU scheme.

The choice of the fingerprint influences the size of the entry and the number of CPU cycles required to compute it. By leveraging processor hardware assists (for e.g., Intel AES [14]) to compute stronger fingerprints (like SHA-2, SHA-256 [28], etc.), the CPU overhead can be greatly mitigated. However, longer, 256-bit fingerprints increase the size of each entry. In addition, a DBN of 32-bits to 64-bits must also be kept within the entry, thus making the minimum entry size 36 bytes. Given a block size of 4 KB (typical of many file systems), the cache entries comprise an overhead of 0.8% of the total size. Since we keep the cache in memory, this overhead is significant as it reduces the number of cached blocks.

In many storage systems, memory not reserved for data structures is used by the buffer cache. Hence, the memory used by the dedup-metadata cache comes at the expense of a larger buffer cache. Therefore, the effect of the dedup-metadata cache on the buffer cache hit ratio needs to be evaluated empirically to size the cache.

3.4.2 Duplicate sequence processing

This subsection describes some common design issues in duplicate sequence identification.

Sequence identification: The goal is to identify the largest sequence among the list of potential sequences. This can be done in multiple ways:

- Breadth-first: Start by scanning blocks in order; concurrently track all possible sequences; and decide on the largest when a sequence terminates.
- Depth-first: Start with a sequence and pursue it across the blocks until it terminates; make multiple passes until all sequences are probed; and then pick the largest. Information gathered during one pass can be utilized to make subsequent passes more efficient.

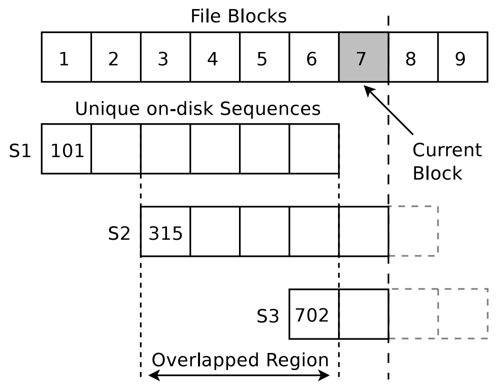


Figure 3: *Overlapped sequences.* This figure shows an example of how the algorithm works with overlapped sequences.

In practice, we observed long chains of blocks during processing (order of 1000s). Since multiple passes is too expensive, we use the breadth-first approach.

Overlapped sequences: Choosing between a set of overlapped sequences can prove problematic. An example of how overlapping sequences are handled is illustrated in Figure 3. Assume a threshold of 4. Scanning from left to right, multiple sequences match the set of file blocks. As we process the 7th block, one of the sequences terminates (S1) with a length 6. But, sequences S2 and S3 have not yet terminated and have blocks overlapping with S1. Since S1 is longer than the threshold (4), we can deduplicate the file blocks matching those in S1. However, by accepting S1, we are rejecting the overlapped blocks from S2 or S3; this is the dilemma. It is possible that either S2 or S3 could potentially lead to a longer sequence going forward, but it is necessary to make a decision about S1. Since it is not possible to know the best outcome, we use the following heuristic: we determine if the set of non-overlapped blocks is greater than threshold, if so, we deduplicate them. Otherwise, we defer to the unterminated sequences, as they may grow longer. Thus, in the example, we reject S1 for this reason.

3.4.3 Threshold determination

The minimum sequence threshold is a workload property that can only be derived empirically. The ideal threshold is one that most closely matches the workload's natural sequentiality. For workloads with more random IO, it is possible to set a lower threshold because deduplication should not worsen the fragmentation. It is possible to have a real-time, adaptive scheme that sets the threshold based on the randomness of the workload. Although valuable, this investigation is beyond this paper's scope.

4 Implementation

In this section, we present the implementation and optimizations of our inline deduplication system. The implementation consists of two subsystems: i) the dedup-metadata management; and ii) the iDedup algorithm.

4.1 Dedup-metadata management

The dedup-metadata management subsystem is comprised of several components:

1. Dedup-metadata cache (in RAM): Contains a pool of block entries (*content-nodes*) that contain deduplication metadata organized as a cache.
2. Fingerprint hash table (in RAM): This table maps a fingerprint to DBN(s).
3. DBN hash table (in RAM): This table maps a DBN to its content-node; used to delete a block.
4. Reference count file (on disk): Maintains reference counts of deduplicated file system blocks in a file.

We explore each of them next.

4.1.1 Dedup-metadata cache

This is a fixed-size pool of small entries called content-nodes, managed as an LRU cache. The size of this pool is configurable at compile time. Each content-node represents a single disk block and is about 64 bytes in size. The content-node contains the block's DBN (a 4 B integer) and its fingerprint. In our prototype, we use the MD5 checksum (128-bit) [33] of the block's contents as its fingerprint. Using a stronger fingerprint (like SHA-256) would increase the memory overhead of each entry by 25%, thus leading to fewer blocks cached. Other than this effect, using MD5 is not expected to alter other experimental results.

All the content-nodes are allocated as a single global array. This allows the nodes to be referenced by their array index (a 4 byte value) instead of by a pointer. This saves 4 bytes per pointer in 64-bit systems. Each content-node is indexed by three data structures: the fingerprint hash table, the DBN hash table and the LRU list. This adds two pointers per index (to doubly link the nodes in a list or tree), thus totaling six pointers per content-node. Therefore, by using array indices instead of pointers we save 24 bytes per entry (37.5%).

4.1.2 Fingerprint hash table

This hash table contains content-nodes indexed by their fingerprint. It enables a block's duplicates to be identified by using the block's fingerprint. As shown in Figure 4, each hash bucket contains a single pointer to the root of a red-black tree containing the collision list for that bucket. This is in contrast to a traditional hash table with a doubly linked list for collisions at the cost of two pointers per bucket. The red-black tree implementation is an optimized, left-leaning, red-black tree [12].

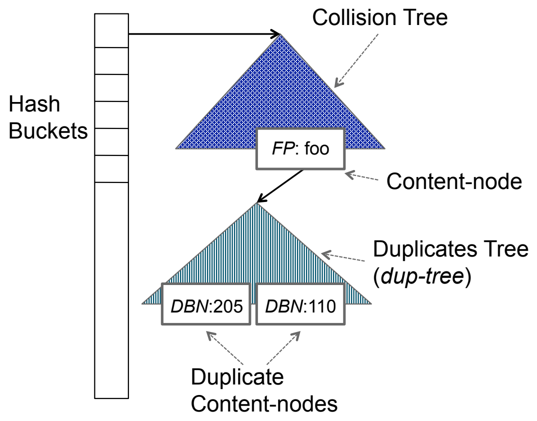


Figure 4: *Fingerprint Hash Table.* The fingerprint hash table with hash buckets as pointers to collision trees. Content-node with fingerprint 'foo' has duplicate content-nodes in a tree (dup-tree) with DBNs 205 and 110.

With uniform distribution, each hash bucket is designed to hold 16 entries, ensuring an upper-bound of 4 searches within the collision tree (tree search cost is O(logN)). By reducing the size of the pointers and the number of pointers per bucket, the per-bucket overhead is reduced, thus providing more buckets for the same memory size.

Each collision tree content-node represents a unique fingerprint value in the system. For thresholds greater than one, it is possible for multiple DBNs to have the same fingerprint, as they can belong to different duplicate sequences. Therefore, all the content-nodes that represent duplicates of a given fingerprint are added to another red-black tree, called the *dup-tree* (see Figure 4). This tree is rooted at the first content-node that maps to that fingerprint. There are advantages to organizing the duplicate content-nodes in a tree, as explained in the iDedup algorithm section (Section 4.2).

4.1.3 DBN hash table

This hash table indexes content-nodes by their DBNs. Its structure is similar to the fingerprint hash table without the dup-tree. It facilitates the deletion of content-nodes when the corresponding blocks are removed from the system. During deletion, blocks can only be identified by their DBNs (otherwise the data must be read and hashed). The DBN is used to locate the corresponding content-node and delete it from all dedup-metadata.

4.1.4 Reference count file

The *refcount file* stores the reference counts of all deduplicated blocks on disk. It is ordered by DBN and maintains a 32-bit counter per block. When a block is deleted, its entry in the refcount file is decremented. When the reference count reaches zero, the block's content-node is removed from all dedup-metadata and the block is

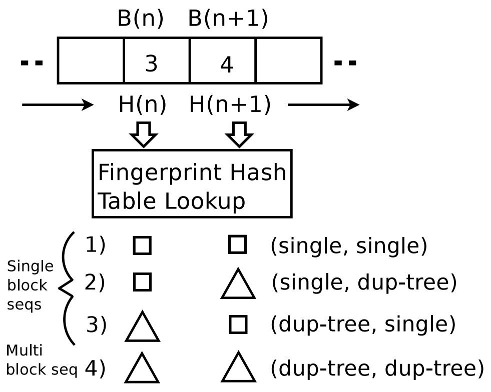


Figure 5: *Identification of sequences.* This figure shows how sequences are identified.

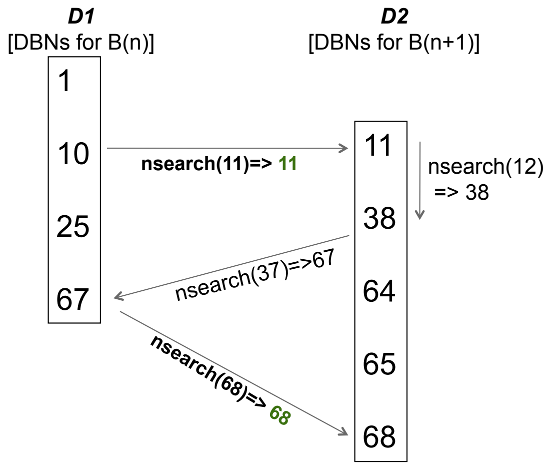


Figure 6: *Sequence identification example.* Sequence identification for blocks with multiple duplicates. D1 represents the dup-tree for block B(n) and D2 the dup-tree for B(n+1).

marked free in the file system's metadata. The refcount file is also updated when a block is written. By deduplicating sequential blocks, we observe that refcount updates are often collocated to the same disk blocks, thereby amortizing IOs to the refcount file.

4.2 iDedup algorithm

For each file, the iDedup algorithm has three phases:

1. Sequence identification: Identify duplicate block sequences for file blocks.
2. Sequence pruning: Process duplicate sequences based on their length.
3. Sequence deduplication: Deduplicate sequences greater than the configured threshold.

We examine these phases next.

4.2.1 Sequence identification

In this phase, a set of newly written blocks, for a particular file, are processed. We use the breadth-first approach for determining duplicate sequences. We start by scanning the blocks in order and utilize the fingerprint hash table to identify any duplicates for these blocks. We filter the blocks to pick only data blocks that are complete (i.e., of size 4 KB) and that do not belong to special or system files (e.g., the refcount file). During this pass, we also compute the MD5 hash for each block.

In Figure 5, the blocks B(n) (n = 1,2,3....) and the corresponding fingerprints H(n) (n = 1,2,3...) are shown. Here, *n* represents the block's offset within the file (the file block number or FBN). The minimum length of a duplicate sequence is two; so, we examine blocks in pairs; i.e., B(1) and B(2) first, B(2) and B(3) next and so on. For each pair, e.g., B(n) and B(n+1) (see Figure 5), we perform a lookup in the fingerprint hash table for H(n) and H(n+1), if neither of them is a match, we allocate the blocks on disk normally and move to the next pair. When we find a match, the matching content-nodes may have more than one duplicate (i.e., a dup-tree) or just a single duplicate (i.e., just an single DBN). Accordingly, to determine if a sequence exists across the pair, we have one of four conditions. They are listed below in increasing degrees of difficulty; they are also illustrated in Figure 5.

1. Both H(n) and H(n+1) match a single content-node: Simplest case, if the DBN of H(n) is b, and DBN of H(n+1) is (b+1), then we have a sequence.
2. H(n) matches a single content-node, H(n+1) matches a dup-tree content-node: If the DBN of H(n) is b; search for (b+1) in the dup-tree of H(n+1).
3. H(n) matches a dup-tree, H(n+1) matches a single content-node: Similar to the previous case with H(n) and H(n+1) swapped.
4. Both H(n) and H(n+1) match dup-tree content-nodes: This case is the most complex and can lead to multiple sequences. It is discussed in greater detail below.

When both H(n) and H(n+1) match entries with dup-trees, we need to identify all possible sequences that can start from these two blocks. The optimized red-black tree used for the dup-trees has a search primitive, `nsearch(x)`, that returns 'x' if 'x' is found; or the next largest number after 'x'; or error if 'x' is already the largest number. The cost of `nsearch` is the same as that of a regular tree search (O(log N)). We use this primitive to quickly search the dup-trees for all possible sequences. This is illustrated via an example in Figure 6.

In our example, we show the dup-trees as two sorted list of DBNs. First, we compute the minimum and maximum overlapping DBNs between the dup-trees (i.e., 10 and 68 in the figure), all sequences will be within this range. We start with 10, since this is in *D1*, the dup-tree of H(n). We then perform a `nsearch(11)` in *D2*, the dup-tree of H(n+1), which successfully leads to a sequence. Since the numbers are ordered, we perform a `nsearch(12)` in D2 to find the next largest potential sequence number; the result is 38. Next, to pair with 38, we perform `nsearch(37)` in D1. However, it results in 67 (not a sequence). Similarly, since we obtained 67 in D1, we perform `nsearch(68)` in D2, thus, yielding another sequence. In this fashion, with a minimal number of searches using the `nsearch` primitive, we are able to glean all possible sequences between the two blocks.

It is necessary to efficiently record, manage, and identify the sequences that are growing and those that have terminated. For each discovered sequence, we manage it via a *sequence entry*: the tuple ⟨Last FBN of sequence, Sequence Size, Last DBN of sequence⟩. Suppose, B(n) and B(n+1) have started a sequence with sequence entry S1. Upon examining B(n+1) and B(n+2), we find that S1 grows and a new sequence, S2, is created. In such a scenario, we want to quickly search for S1 and update its contents and create a new entry for S2. Therefore, we maintain the sequence entries in a hash table indexed by a combination of the tuple fields. In addition, as we process the blocks, to quickly determine terminated sequences, we keep two lists of sequence entries: one for sequences that include the current block and another for sequences of the previous block. After sequence identification for a block completes, if a sequence entry is not in the current block's list, then it has terminated.

4.2.2 Sequence pruning

Once we determine the sequences that have terminated, we process them according to their sizes. If a sequence is larger than the threshold, we check for overlapping blocks with non-terminated sequences using the heuristic mentioned in Section 3.4.2, and only deduplicate the non-overlapped blocks if they form a sequence greater than the threshold. For sequences shorter than the threshold, the non-overlapped blocks are allocated by assigning them to new blocks on disk.

4.2.3 Deduplication of blocks

For each deduplicated block, the file's metadata is updated with the original DBN at the appropriate FBN location. The appropriate block in the refcount file is retrieved (a potential disk IO) and the reference count of the original DBN is incremented. We expect the refcount updates to be amortized across the deduplication of multiple blocks for long sequences.

5 Experimental evaluation

In this section, we describe the goals of our evaluation followed by details and results of our experiments.

5.1 Evaluation objectives

Our goal is to show that a reasonable tradeoff exists between performance and deduplication ratio that can be exploited by iDedup for latency sensitive, primary workloads. In our system, the two major tunable parameters are: i) the minimum duplicate sequence threshold, and ii) the in-memory dedup-metadata cache size. Using these paramaters we evaluate the system by replaying traces from two real-world, enterprise workloads to examine:

1. Deduplication ratio vs. threshold: We expect a drop in deduplication rate as threshold increases.
2. Disk fragmentation profile vs. threshold: We expect the fragmentation to decrease as threshold increases.
3. Client read response time vs. threshold: We expect the client read response time characteristics to follow the disk fragmentation profile.
4. System CPU utilization vs. threshold: We expect the utilization to increase slightly with the threshold.
5. Buffer cache hit rate vs. dedup-metadata cache size: We expect the buffer cache hit ratio to decrease as the metadata cache size increases.

We describe these experiments and their results next.

5.2 Experimental setup

All evaluation is done using a NetApp® FAS 3070 storage system running Data ONTAP® 7.3 [27]. It consists of: 8 GB RAM; 512 MB NVRAM; 2 dual-core 1.8 GHz AMD CPUs; and 3 10K RPM 144 GB FC Seagate Cheetah 7 disk drives in a RAID-0 stripe. The trace replay client has a 16-core, Intel® Xeon® 2.2 GHz CPU with 16 GB RAM and is connected by a 1 Gb/s network link.

We use two, real-world, CIFS traces obtained from a production, primary storage system that was collected and made available by NetApp [20]. One trace contains Corporate departments' data (MS Office, MS Access, VM Images, etc.), called the *Corporate* trace; it contains 19,876,155 read requests (203.8 GB total read) and 3,968,452 write requests (80.3 GB total written). The other contains Engineering departments' data (user home dirs, source code, etc.), called the *Engineering* trace; it contains 23,818,465 read requests (192.1 GB total read) and 4,416,026 write requests (91.7 GB total written). Each trace represents ≈ 1.5 months of activity. They are replayed without altering their data duplication patterns.

We use three dedup-metadata cache sizes: 1 GB, 0.5 GB and 0.25 GB, that caches block mappings for approximately 100%, 50% and 25% of all blocks written in the trace respectively. For the threshold, we use reference values of 1, 2, 4, and 8. Larger thresholds produce insignificant deduplication savings to be feasible.

Two key comparison points are used in our evaluation:

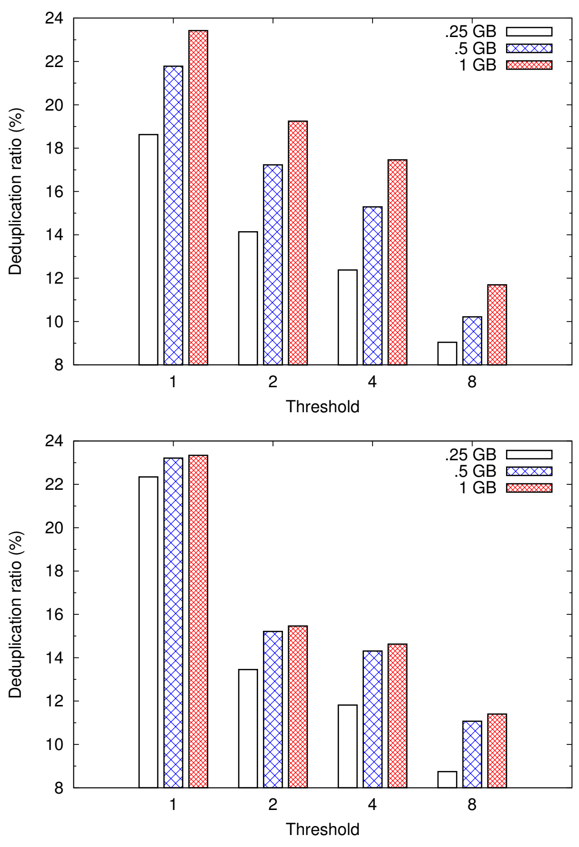


Figure 7: *Deduplication ratio vs. Threshold.* Deduplication ratio versus threshold for the different cache sizes for Corporate (top) and Engineering (bottom) traces.

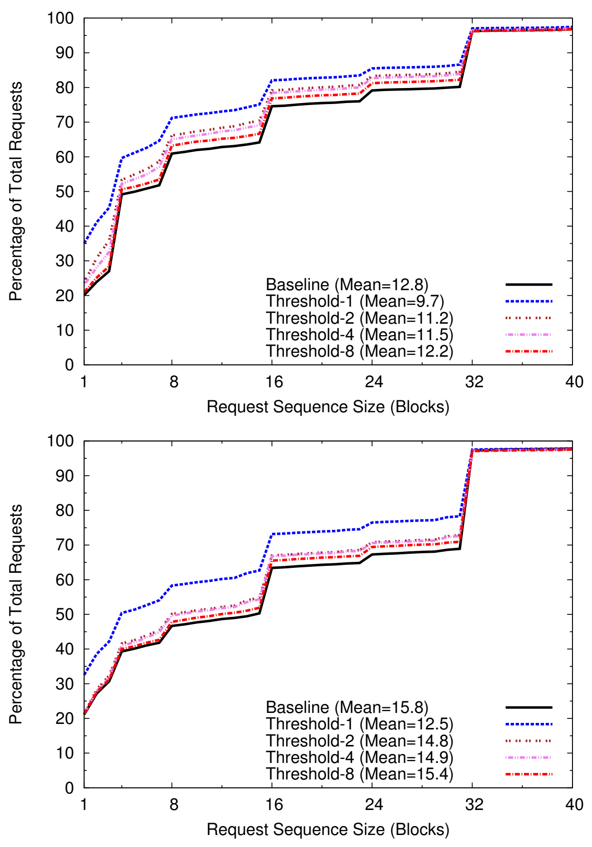


Figure 8: *Disk fragmentation profile.* CDF of number of sequential blocks in disk read requests for the Corporate (top) and Engineering (bottom) traces with a 1G cache.

1. The *Baseline* values represent the system without the iDedup algorithm enabled (i.e., no deduplication).
2. The Threshold-1 values represent the highest deduplication ratio for a given metadata cache size. Since a 1 GB cache caches all block mappings, Threshold-1 at 1 GB represents the maximum deduplication possible (with a 4 KB block size) and is equivalent to a static offline technique.

5.3 Deduplication ratio vs. threshold

Figure 7 shows the tradeoff in deduplication ratio (dedup-rate) versus threshold for both the workloads and different dedup-metadata sizes. For both the workloads, as the threshold increases, the number of duplicate sequences decrease, correspondingly the dedup-rate drops; there is a 50% decrease between Threshold-1 (24%) and 8 (13%), with a 1 GB cache. Our goal is to maximize the size of the threshold, while also maintaining a high dedup-rate. To evaluate this tradeoff, we look for a range of useful thresholds (> 1) where the drop in dedup-rate is not too steep; e.g., the dedup-rates between Threshold-2 and Threshold-4 are fairly flat. To minimize performance impact, we would pick the largest threshold that shows the smallest loss in dedup-rate: Threshold-4 from either graph. Moreover, we notice the drop in dedup-rate from Threshold-2 to Threshold-4 is same for 0.5 GB and 0.25 GB ($\approx$ 2%), showing a bigger percentage drop for smaller caches. For the Corporate workload, iDedup achieves a deduplication ratio between 66% (at Threshold-4, 0.25 GB) and 74% (at Threshold-4, 1 GB) of the maximum possible ($\approx$ 24% at Threshold-1, 1 GB). Similarly, with the Engineering workload, we achieve between 54% (at Threshold-4, 0.25 GB) and 62% (at Threshold-4, 1 GB) of the maximum ($\approx$ 23% at Threshold-1, 1 GB).

5.4 Disk fragmentation profile

To assess disk fragmentation due to deduplication, we gather the number of sequential blocks (request size) for each disk read request across all the disks and plot them as a CDF (cumulative distribution function). All CDFs are based on the average over three runs. Figure 8 shows the CDFs for both Corporate and Engineering workloads for a dedup-metadata cache of 1 GB. Other cache sizes show similar patterns. Since the request stream is the same for all thresholds, the difference in disk IO sizes, across the different thresholds, reflects the fragmentation of the file system's disk layout.

As expected, in both the CDFs, the Baseline shows the highest percentage of longer request sizes or sequentiality; i.e., the least fragmentation. Also, it can observed that the Threshold-1 line shows the highest amount of fragmentation. For example, there is a 11% increase in the number of requests smaller or equal to 8, between the Baseline and Threshold-1 for the Corporate workload and 12% for the Engineering workload. All the remaining thresholds (2, 4, 6, 8) show progressively less fragmentation, and have CDFs between the Baseline and the Threshold-1 line; e.g., a 2% difference between Baseline and Threshold-8 for the Corporate workload. Hence, to optimally choose a threshold, we suggest the tradeoff is made after empirically deriving the dedup-rate graph and the fragmentation profile. In the future, we envision enabling the system to automatically make this tradeoff.

5.5 Client response time behavior

Figure 9 (top graph) shows a CDF of client response times taken from the trace replay tool for varying thresholds of the Corporate trace at 1 GB cache size. We use response time as a measure of latency. For thresholds of 8 or larger, the behavior is almost identical to the Baseline (an average difference of 2% for Corporate and 4% for Engineering at Threshold 8) , while Threshold-2 and 4 (not shown) fall in between. We expect the client response time to reflect the fragmentation profile. However, the impact on client response time is lower due to the storage system's effective read prefetching.

As can be seen, there is a slowly shrinking gap between Threshold-1 and Baseline for larger response times (> 2ms) comprising $\approx$ 10% of all requests. The increase in latency of these requests is due to the fragmentation effect and it affects the average response time. To quantify this better, we plot the difference between the two curves in the CDF (bottom graph of Figure 9) against the response time. The area under this curve shows the total contribution to latency due to the fragmentation effect. We find that it adds 13% to the average latency and a similar amount to the total runtime of the workload, which is significant. The Engineering workload has a similar pattern, although the effect is smaller (1.8% for average latency and total runtime).

5.6 System CPU utilization vs. threshold

We capture CPU utilization samples every 10 seconds from all the cores and compute the CDF for these values. Figure 10 shows the CDFs for our workloads with a

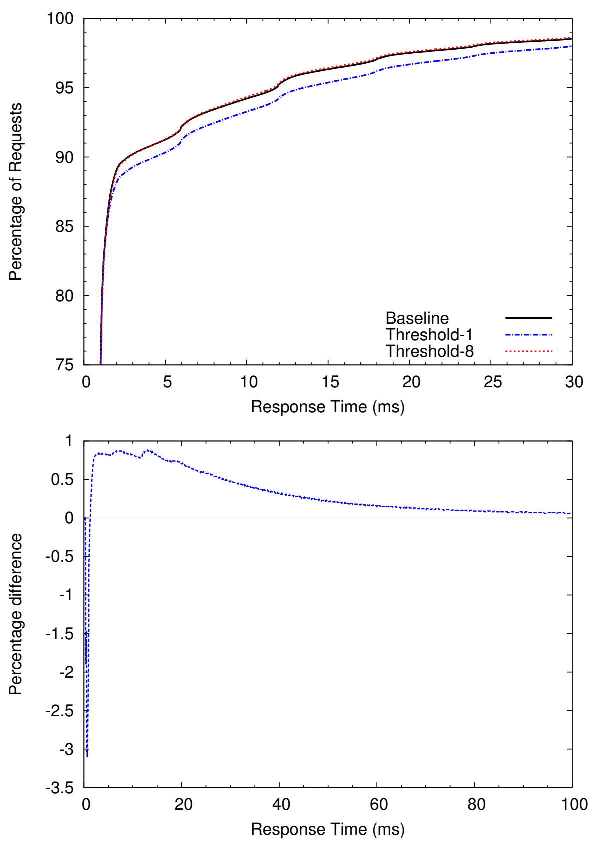
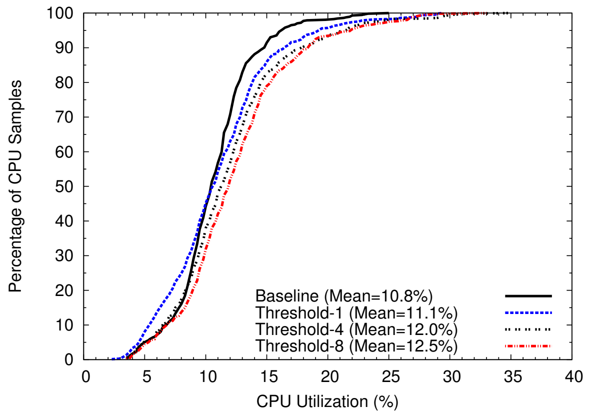


Figure 9: *Client response time CDF.* CDF of client response times for Corporate with a 1 GB cache (top); we highlight the region where the curves differ. The difference between the Baseline and Threshold of 1 CDFs (bottom).

1 GB dedup-metadata cache. We expect Threshold-8 to consume more CPU because there are potentially more outstanding, unterminated sequences leading to more sequence processing and management. As expected, compared to the Baseline, the maximum difference in mean CPU utilization occurs at Threshold-8, but is relatively small: $\approx$ 2% for Corporate and $\approx$ 4% for Engineering. However, the CDFs for the thresholds exhibit a longer tail, implying a larger standard deviation compared to the Baseline, this is evident in the Engineering case but less so for Corporate. However, given that the change is small (< 5%), we feel that the iDedup algorithm has little impact on the overall utilization. The results are similar across cache sizes, we chose the maximal 1 GB one, since that represents maximum work in sequence processing for the iDedup algorithm.

5.7 Buffer cache hit ratio vs. metadata size

We observed the buffer cache hit ratio for different sizes of the dedup-metadata cache. The size of the dedup-metadata cache (and threshold) had no observable ef-



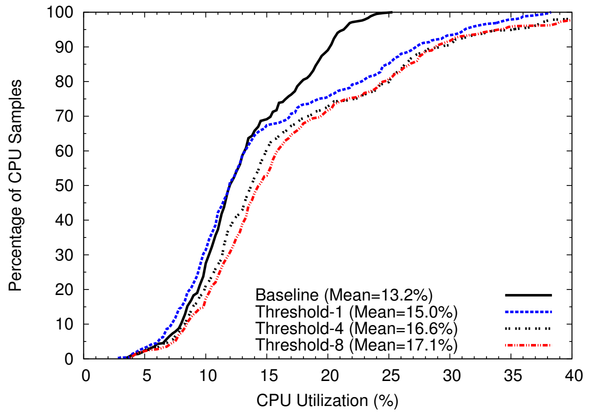


Figure 10: *CPU Utilization CDF.* CDF across all the cores for varying thresholds for Corporate (top) and Engineering (bottom) workloads with a 1 GB cache. Threshold-2 is omitted, since it almost fully overlaps Threshold-4.

fect on the buffer cache hit ratio for two reasons: i) the dedup-metadata cache size (max of 1 GB) is relatively small compared to the total memory (8 GB); and ii) the workloads' working sets fit within the buffer cache. The buffer cache hit ratio was steady for the Corporate (93%) and Engineering (96%) workloads. However, workloads with working sets that do not fit in the buffer cache would be impacted by the dedup-metadata cache.

6 Related work

Data storage efficiency can be realized via various complementary techniques such as thin-provisioning (not all of the storage is provisioned up front), data deduplication, and compression. As shown in Table 1 and as described in Section 2, deduplication systems can be classified as primary or secondary (backup/archival). Primary storage is usually optimized for IOPs and latency whereas secondary storage systems are optimized for throughput. These systems either process duplicates inline, at ingest time, or offline, during idle time.

Another key trade-off is with respect to the deduplication granularity. In file level deduplication (e.g., [18, 21, 40]), the potential gains are limited compared to deduplication at block level. Likewise, there are algorithms for fixed-sized block or variable-sized (e.g., [4, 23]) block deduplication. Finally, there are content addressable systems (CAS) that reference the object or block directly by its content hash; inherently deduplicating them [24, 31].

Although, we are unaware of any prior primary, inline deduplication systems, offline systems do exist. Some are block-based [1, 16], while others are file-based [11].

Complementary research has been done on inline compression for primary data [6, 22, 38]. Burrows et. al [5] describe an on-line compression technique for primary storage using a log-structured file system. In addition, offline compression products also exist [29].

The goals for inline secondary or backup deduplication systems are to provide high throughput and high deduplication ratio. Therefore, to reduce the amount of in-memory dedup-metadata footprint and the number of metadata IOs, various optimizations have been proposed [2, 15, 21, 39, 41]. Another inline technique, by Lillibridge et al. [21], leverages temporal locality with sampling to reduce dedup metadata size in the context of backup streams.

Deduplication systems have also leveraged flash storage to minimize the cost of metadata IOs [7, 25]. Clustered backup storage systems have been proposed for large datasets that cannot be managed by a single backup storage node [8].

7 Conclusion

In this paper, we describe iDedup, an inline deduplication system specifically targeting latency-sensitive, primary storage workloads. With latency sensitive workloads, inline deduplication has many challenges: fragmentation leading to extra disk seeks for reads, deduplication processing overheads in the critical path, and extra latency caused by IOs for dedup-metadata management.

To counter these challenges, we derived two insights by observing real-world, primary workloads: i) there is significant spatial locality on disk for duplicated data, and ii) temporal locality exists in the accesses of duplicated blocks. First, we leverage spatial locality to perform deduplication only when the duplicate blocks form long sequences on disk, thereby, avoiding fragmentation. Second, we leverage temporal locality by maintaining dedup-metadata in an in-memory cache to avoid extra IOs. From our evaluation, we see that iDedup offers significant deduplication with minimal resource overheads (CPU and memory). Furthermore, with careful threshold selection, a good compromise between performance and deduplication can be reached, thereby, making iDedup well suited to latency sensitive workloads.

References

[1] C. Alvarez. NetApp deduplication for FAS and V-Series deployment and implementation guide. Technical Report TR-3505, NetApp, 2011.

[2] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, pages 1–9, Sept. 2009.

[3] J. Bonwick. Zfs deduplication. `http://blogs.oracle.com/bonwick/entry/zfs_dedup`, Nov. 2009.

[4] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398–409, May 1995.

[5] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–9, Boston, MA, Oct. 1992.

[6] C. Constantinescu, J. S. Glider, and D. D. Chambliss. Mixing deduplication and compression on active data sets. In *Proceedings of the 2011 Data Compression Conference*, pages 393–402, Mar. 2011.

[7] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 16–16, June 2010.

[8] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST)*, pages 15–29, Feb. 2011.

[9] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST)*, pages 197–210, Feb. 2009.

[10] L. DuBois, M. Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. White Paper 223310, Mar. 2011.

[11] EMC. Achieving storage efficiency through EMC Celerra data deduplication. White paper, Mar. 2010.

[12] J. Evans. Red-black tree implementation. `http://www.canonware.com/rb`, 2010.

[13] GreenBytes. GreenBytes, GB-X series. `http://www.getgreenbytes.com/products`.

[14] S. Gueron. Intel advanced encryption standard (AES) instructions set. White Paper, Intel, Jan. 2010.

[15] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 25–25, June 2011.

[16] IBM Corporation. IBM white paper: IBM Storage Tank – A distributed storage system, Jan. 2002.

[17] IDC. The 2011 digital universe study. Technical report, June 2011.

[18] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST)*, pages 281–294, Dec. 2005.

[19] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the Eight USENIX Conference on File and Storage Technologies (FAST)*, pages 211–224, Feb. 2010.

[20] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 213–226, June 2008.

[21] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST)*, pages 111–123, Feb. 2009.

[22] LSI. LSI WarpDrive SLP-300. `http://www.lsi.com/products/storagecomponents/Pages/WarpDriveSLP-300.aspx`, 2011.

[23] U. Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 1–10, Jan. 1994.

[24] T. McClure and B. Garrett. EMC Centera: Optimizing archive efficiency. Technical report, Jan. 2009.

[25] D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (ssd). In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, pages 1–6, June 2010.

[26] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST)*, pages 1–13, Feb. 2011.

[27] Network Appliance Inc. Introduction to Data ONTAP 7G. Technical Report TR 3356, Network Appliance Inc.

[28] NIST. Secure hash standard SHS. Federal Information Processing Standards Publication FIPS PUB 180-3, Oct. 2008.

[29] Ocarina. Ocarina networks. `http://www.ocarinanetworks.com/`, 2011.

[30] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

[31] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[32] S. C. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 143–156, June 2008.

[33] R. Rivest. The MD5 message-digest algorithm. Request For Comments (RFC) 1321, IETF, Apr. 1992.

[34] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[35] J. Satran. Internet small computer systems interface (iSCSI). Request For Comments (RFC) 3720, IETF, Apr. 2004.

[36] S. Silverberg. `http://opendedup.org`, 2011.

[37] Storage Networking Industry Association. Common Internet File System (CIFS) Technical Reference, 2002.

[38] StorWize. Preserving data integrity assurance while providing high levels of compression for primary storage. White paper, Mar. 2007.

[39] W. Xia, H. Jiang, D. Feng, and Y. 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*, pages 26–28, June 2011.

[40] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, Apr. 2005. IEEE.

[41] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST)*, pages 269–282, Feb. 2008.