DEDUPLICATION-AWARE PAGE CACHE IN LINUX KERNEL FOR IMPROVED READ PERFORMANCE

by

VENKATA SATYA RAVI KIRAN BOGGAVARAPU

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2019

To my parents

ACKNOWLEDGMENTS

I would like to thank Dr. Song Jiang, my thesis advisor, for his invaluable guidance and support through my master thesis. I am thankful to thank Dr. Jia Rao and Dr. Mohammad Atiqul Islam, for accepting to be on my thesis committee. I sincerely appreciate the efforts of Dr. Fan Ni for his remarkable guidance in this project.

I thank my cousins Ravi, Anil, and Sairam for supporting me through my master’s degree. I thank my friend Sravan for his help and support.

I am grateful to my parents for all their unconditional love and support. They sacrificed a lot to provide me a good education. I am lucky to have such a progressive and understanding parents. I thank my pet dog, Robin, for being with my parents for the past nine years and keeping them busy. I thank Lavanya, the love of my life, for supporting me in whatever I choose to pursue. I am so grateful to have her unconditional love.

December 11, 2019

ABSTRACT

DEDUPLICATION-AWARE PAGE CACHE IN LINUX KERNEL FOR IMPROVED READ PERFORMANCE

VENKATA SATYA RAVI KIRAN BOGGAVARAPU, MS

THE UNIVERSITY OF TEXAS AT ARLINGTON, 2019

Supervising Professor: Song Jiang

The amount of data being produced and consumed is increasing every day. As a result, there can be a large amount of redundant data in the storage system. Storing and accessing these duplicate data unnecessarily consumes disk space and I/O bandwidth. Deduplication techniques are widely deployed to remove the redundancy. In particular, the deduplication solutions that work at the block level are proven to be effective. These solutions aim to effectively use disk space and write bandwidth by avoiding duplicate data writes to the storage. However, such a design might not help in improving the read performance, which is critical for many modern-day applications.

The Linux kernel implements an in-memory cache of pages, called the page cache, to improve I/O performance by minimizing disk accesses. The page cache has pages originating from regular file systems, and it is indexed by a file and the offset within the file. However, due to such a design, deduplication information is currently not available to the page cache. Due to this, the kernel cannot avoid read requests from going to the disk on offsets that are not present in the page cache, even though the requested data duplicates another offset that is already cached. Consequently, the overall I/O performance of the applications running on these systems can be compromised.

To address this issue, we propose a lightweight scheme called Dual-Dedup, that efficiently coordinates the deduplication information with the page cache. It discloses the redundancy knowledge detected by the block-level deduplication layer to the page cache, which can then prevent unnecessary read requests. Results from extensive experiments show that Dual-Dedup significantly improves read performance. On FIO tests with 25% duplicate data, our system shows an improvement of 34% in the read throughput when compared with Linux EXT4.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

In this big data era, there is a paradigm shift towards data-intensive computing. Two characteristics define this shift: 1) there is an increase in the number of applications that deal with vast data, and 2) I/O utilization started to become the most valuable resource metric. For example, big data analytics systems rely on storage systems that provide faster access to data and high space efficiency to deliver their results in time. As the amount of data produced and consumed is increasing every day, over time, the storage system may contain a large amount of redundant data [1][2][6][7]. Examples include the same or similar audio/video/image files uploaded by multiple users and versions of a data set with minor differences between them. This redundant data not only consumes unnecessary storage space but also eats up the valuable I/O bandwidth.

Data deduplication techniques have been widely deployed to remove the redundancy. In particular, the deduplication solutions that work at the block level are proven to be effective [1]. Block-level deduplication solutions work on the granularity of blocks and are designed to be flexible with the file system. These solutions aim to effectively utilize disk space and write bandwidth by avoiding duplicate data writes to the storage. However, such a design might not help in improving the read performance, which is critical for many modern-day applications.

A read operation mostly sits on the critical path of a program's execution. For a program to run, it must be first read from the storage (often slower devices such as hard disks and SSDs) and brought into the memory (e.g., DRAM). Also, during their lifetime, most programs access and modify data. Therefore, the impact of read operation on the overall user-perceived I/O performance is higher than writes most of the time. The operating system (OS) implements an in-memory cache to improve the I/O performance. In the Linux kernel, this cache is called the page cache, which is a cache for all pages originating from regular disk-based file systems. The page cache is a system-wide data structure indexed by a file and the offset within the file. Due to this design, there is no way for the page cache to find if the data contents at two different offsets are the same. Therefore, the page cache will allow an unnecessary read request to be passed down to the disk, even if a duplicate copy of the same data is present at a different offset. Thus, the overall I/O performance of data-intensive applications running on these systems can be compromised.

A straightforward solution to this issue can be obtained by caching unique data blocks in the memory [2]. Although this solution looks convincing, because of the greater control we get on the cache, it has its disadvantages. Memory is a resource of a smaller capacity, more performance-critical, and much more expensive. Therefore, we need to limit the size of the cache. If not, over time, the cache can occupy the entire memory, leaving no space for the applications to run. Now to improve the utilization of a cache in limited memory, we need to have effective cache replacement policies. Finally, even if we do keep a fixed-size cache with efficient replacement policies, we will be wasting valuable memory by storing redundant data in it. This is because the operating system will ultimately cache the data, in its page cache, during reads and writes. Therefore, creating a separate cache is not a viable solution.

A much more efficient solution to this issue can be obtained by integrating deduplication information with the page cache. In this thesis, we propose a scheme called Dual-Dedup, which does that. Dual-Dedup is a lightweight scheme that maps physical locations of unique data blocks on disk with the corresponding page cache entries. Our design goal is to make Dual-Dedup extremely lightweight, both in terms of memory and I/O overhead.

Memory accesses are much faster than disk operations. Accordingly, in-memory deduplication operations must match the speed of user programs' memory access speed. However, redundancy detection requires running expensive cryptographic hash functions, such as SHA-1 or MD5, to calculate fingerprints of data blocks and to search for matched fingerprints. These operations can add an excessive cost on the read performance. Fortunately, the block-layer deduplication solution has always paid the cost for detecting the redundancy. In the design, we will manage to share the knowledge gained at this layer with the page cache, so that the page cache can conduct its deduplication almost for free. Therefore, we name the proposed design Dual-Dedup to highlight the synergy between the deduplication subsystem, running at the block level, and the page cache to collectively support high-performance applications running on top of them.

CHAPTER 2
TECHNICAL BACKGROUND

Before going into the details of our system, let us look at some of the required technical background necessary. In this chapter, we first discuss the anatomy of the read system call in the Linux kernel, in terms of the layers in the kernel's I/O stack. We then look at the design of the page cache. Finally, we give a brief introduction to deduplication and the design of Dmdedup.

Anatomy of Read System Call in Linux Kernel I/O Stack

Linux kernel's I/O stack can be logically divided into different layers, as shown in Figure 1. Applications sit at the top of the stack. Block devices, such as disks and SSDs, are located at the bottom. Whatever that is below the application layer is known as the kernel. Linux provides several system call interfaces that allow applications to interact with its kernel [8]. A system call interface acts as a bridge of communication between applications and the operating system. Read system call is used by a program that needs to access data from a file stored in a file system. Let us see how the read system call travels through different layers of the kernel's I/O stack during its lifetime.

A read request issued by an application first goes to the Virtual File System (VFS) layer. VFS provides generic file handling functions that are used by most file systems. A read request in the VFS is first checked in the page cache. The page cache (also known as buffer cache or the VFS cache) is a cache of pages originating from all the file systems.

If the requested offset is present in the page cache, the corresponding page contents are directly copied from the page cache to the application's memory (user-space), marking the completion of the read request. This is called a "hit" in the page cache. However, if the requested offset cannot be found in the page cache, a new page will be allocated and added to the page cache, and the corresponding read request will be passed down to the bottom layers. This is called a "miss" in the page cache.

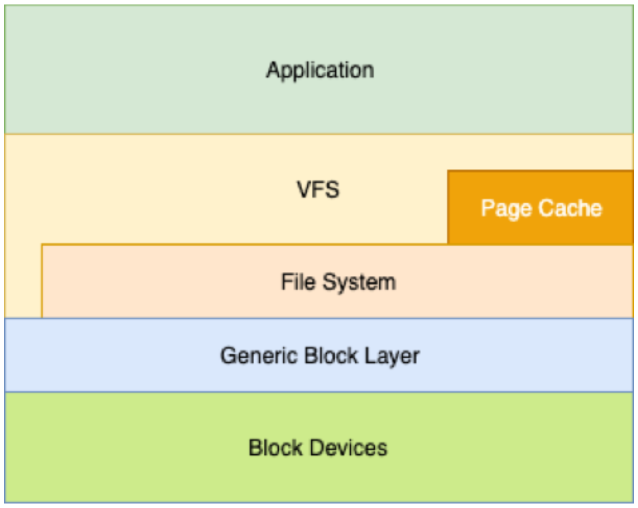

Figure 1   A high-level view of the Linux kernel I/O stack

When the page cache encounters a miss, the read request is passed to the specific file system. The file system determines the corresponding sectors to read the data blocks from the disk. This information is file system specific, as each file system may have its layout of the files. Once the sector numbers are determined, the read request is passed down to the Generic Block Layer.

Like VFS, the generic block layer is aimed to provide a consistent set of functions for various underlying block devices. BIO (Block I/O) is the primary data structure at this layer. Once the Generic Block Layer receives a read request, it creates a new BIO structure and embeds the page allocated by the VFS into the structure. Generic Block Layer also handles the queue for all the pending BIOs. The newly created BIO will be first added to the I/O queue. The BIO will later be passed to the disk driver of the corresponding block device, and finally, to the device to fill the data into the page.

Once the data has been filled into the page, each layer reports completion to the layer on its top. After the file system layer determines that the read has no errors, it informs the VFS layer, which can then copy the data to the user-space. In this way, the page cache stores a copy for itself to make the future accesses to that offset faster.

The Page Cache

In this section, we briefly talk about one of the fundamental caches in a modern computer operating system: the page cache. In Linux, the page cache is a pure software cache for pages originating from all the file systems. The purpose of the page cache is to improve the I/O performance of the operating system by caching frequently accessed files that are stored on the disk in memory. OS maintains the page cache in otherwise unused parts of the memory. Thus, all the available memory will be used for the page cache.

The page cache is a system-wide data structure, implemented as a Radix Tree, and managed per file object. Figure 2 shows an illustration of the page cache tree. A page in the page cache cannot be identified by a device number and a block number because a page does not necessarily contain physically adjacent data blocks. Instead, a page in the page cache is identified by the file and the offset within it. Due to such kind of design, page cache may contain duplicates pages [3]. For instance, consider a case where an offset is read from a file and written to another file. In that case, page cache happily caches the pages from the source as well as destination files.

The important takeaway here is that the memory allocated to the page cache is always considered as available memory. Whenever an application demands additional memory, the pages in the page cache will be evicted to make room for the application.

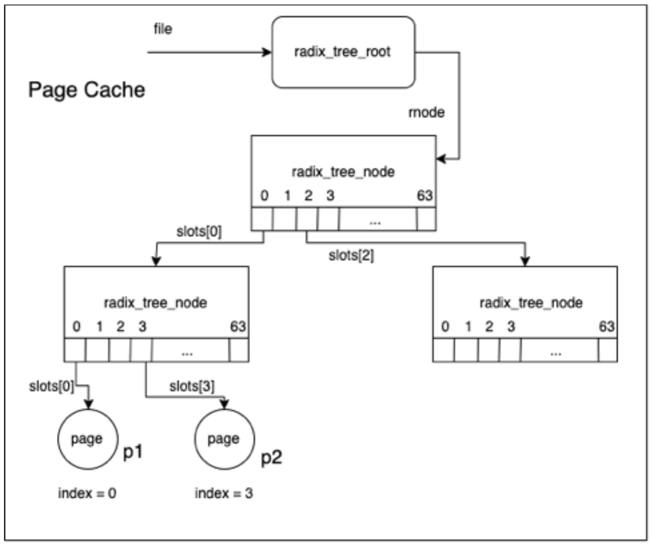

Figure 2 Example of page cache tree implemented as a radix tree

Data Deduplication

Data deduplication (also known as Deduplication, Dedupe, or Dedup) is a technique to remove redundant copies of repeating data. Deduplication saves storage space that would otherwise have wasted by storing redundant copies of data. First, the data set is partitioned into chunks in a phase called chunking [5]. These data chunks will be sent to a cryptographic hash function (e.g., CRC or MD5) to get a hashed output for the data chunk. The hashed output, known as fingerprint, will be unique to that data chunk. Next comes the matching phase, where this fingerprint is compared with existing fingerprints to find a match. If found a match, the data chunk is treated as a duplicate.

There are two forms of deduplication, in-line and offline (post-process). In in-line duplication, the duplicates among the data are detected when the data is written to the disk. Whereas in offline deduplication, the storage is scanned to detect duplicates among stored data.

Once a duplicate data chunk is detected, either it will be prevented from being stored on the disk (in-line) or will be removed if already stored (offline). Instead, a soft link will be created that maps this duplicate data's address to the unique data chunk that exists in the storage.

Dmdedup

Dmdedup is a primary-storage data deduplication platform. It is an in-line block-level deduplication solution that detects redundancy among the data blocks at the block layer of the kernel's I/O stack. The primary data structures in Dmdedup are Hash-PBN mapping and LBN-PBN mapping. Let us see how Dmdedup uses these data structures.

The Hash-PBN table maps the fingerprint of a data block to its corresponding physical location. Therefore, it is a one-to-one mapping. Dmdedup receives a data block that contains data contents and the address to which the application wants to write this data. This address is called the Logical Block Address (LBN).

This data block will go through the chunking phase to get the fingerprint of the data contents. After calculating a data chunk's fingerprint, it will be compared with the existing fingerprints in the Hash-PBN table. If a match is not found, this chunk of data is treated as unique and stored on the disk. At this stage, an entry, with the fingerprint as a key and the location unique of the data chunk on the disk as value, will be inserted into the Hash-PBN mapping.

However, if a fingerprint match is found in the Hash-PBN table, the data chunk is treated as a duplicate copy and not stored on the disk. At this point, the Dmdedup creates an entry, with the LBN as a key, and the location of the unique data chunk on the disk as value will be inserted into the LBN-PBN mapping.

CHAPTER 3
SYSTEM DESIGN

The design goal of Dual-Dedup is to improve the read performance in operating systems by leveraging the knowledge about on-disk duplicates. Block-level deduplication solutions, such as Dmdedup [1], detect the duplicates among the data when they are written to the disk. The deduplication solution maintains the metadata information about duplicate addresses and the corresponding address of their unique data blocks. Dual-Dedup discloses this information about on-disk duplicates, detected by the block-level deduplication solution, to the page cache manager. To this end, we have created a lightweight scheme that maps unique data blocks on the disk with their corresponding page cache entries. In this chapter, we discuss the design and implementation of Dual-Dedup.

Design of Dual-Dedup

As mentioned in Chapter 2, a read request travels through various layers of the Linux kernel's I/O stack during its lifetime. For a system like Dual-Dedup, this means that there is more than one choice of the layer at which a read request, detected as unnecessary, can be intercepted and avoided from passing further down. It brings the question about what the ideal choice of the layer is to implement our system. A system such as Dual-Dedup can be implemented in any of the layers, shown in Figure 1, of the Linux kernel. We chose the VFS layer to implement Dual-Dedup because of the flexibility and compatibility that the layer has to offer.

Before discussing the rationale behind our choice, let us first see what the other choices available are and why they are not good candidates to implement Dual-Dedup. The first one is the application layer, which is the top layer of the I/O stack. The application layer falls in the user-space of the Linux kernel. Since the deduplication information is available at the block layer, which is in the kernel-space, we need to have some mechanism to communicate this information to the application layer. One way to accomplish this is by implementing system calls. However, doing so can harm the performance of the applications. It is because of frequent context switches that are required to access the information in the kernel space from the user level. The other concern with choosing the application layer is that the application programmer should be aware of, and use, the block-level deduplication. It requires modifying all the existing applications that are currently running on the operating system. Therefore, implementing Dual-Dedup at the application level will be impractical.

The second choice is modifying an existing file system. It might seem like a viable choice because we can use the file-system-specific information, but such a design is not flexible as it would be restricted to a specific file system. When the VFS layer finds out that there is a page cache miss, it allocates a page and passes the request to the bottom of the I/O stack to fill the data into the page from the disk. Therefore, if we were to implement our system in the file system layer, we would be allowing the kernel to make unnecessary allocations that might compromise the I/O performance. The same applies to the third available choice (block layer) because the block layer is situated even further down in the I/O stack.

The fourth choice we had considered is an implementation in the page cache itself. However, the page cache is a highly optimized data structure. It is expected to provide fast lookups. So, modifying the program flow of the page cache would not be an ideal choice.

Finally, we decided to implement Dual-Dedup in the VFS layer. The choice of design would be ideal because it will be flexible with the underlying file system, and since VFS is the only layer that interacts with the page cache, our system would be compatible with all the norms of the VFS layer. As mentioned in Chapter 1, Dual-Dedup does not detect duplicates in the data by itself. Instead, it leverages the information provided by the block-level deduplication solution Dmdedup.

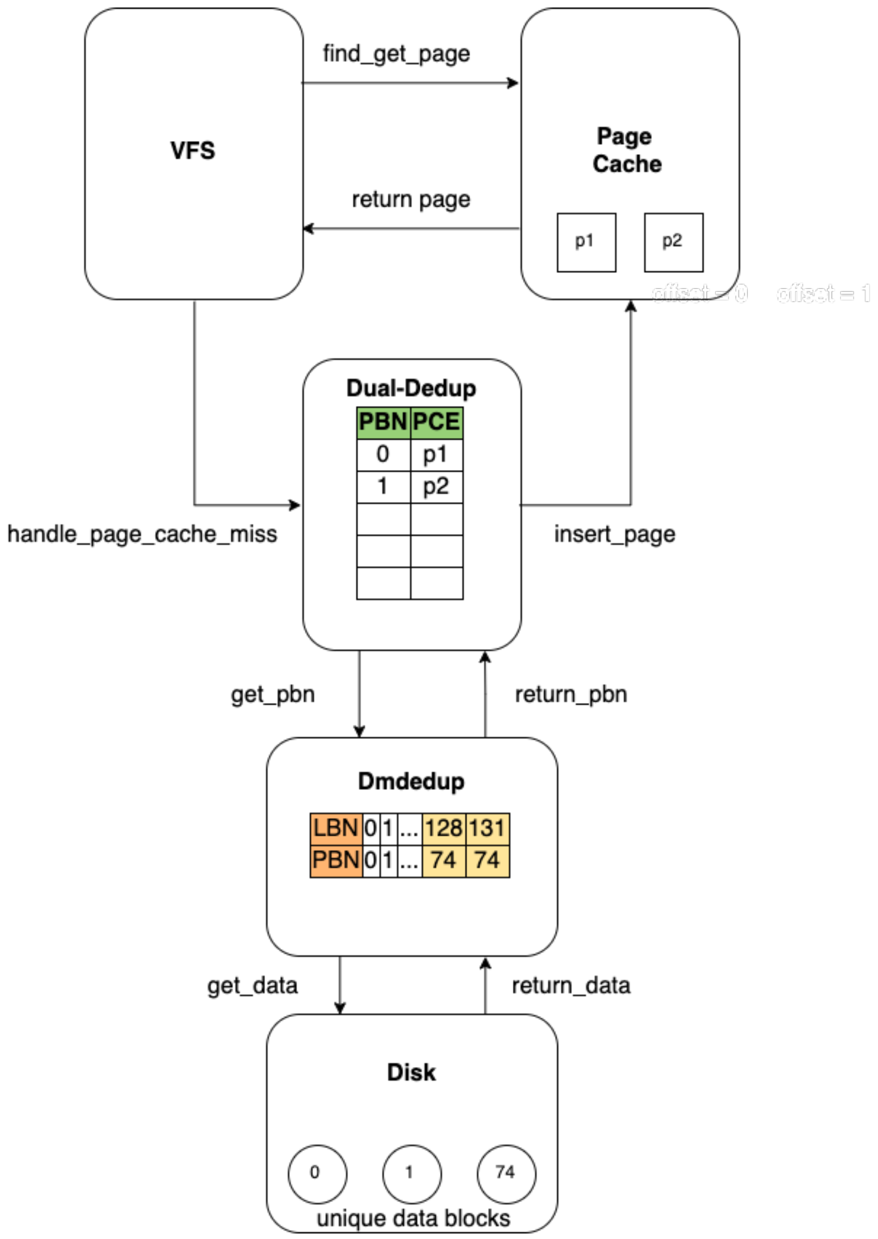

Figure 3 Position of Dual-Dedup in the Linux Kernel I/O stack

<u>Implementation of Dual-Dedup</u>

Once we decide on the layer, now it is the time to think about how to implement Dual-Dedup. Since read operation sits in the critical path of the program's execution, it is essential to minimize the overhead caused by our implementation.

The primary data structure in Dual-Dedup is the PBN-PCE mapping table. It is implemented as a direct-address linear mapping that maps a Physical Block Number (PBN) to its corresponding Page Cache Entry (PCE). An entry in the page cache is identified by the file and the offset inside the corresponding file. Therefore, the PCE consists of these two members. Dual-Dedup is extremely lightweight, as it needs only 16 Bytes for a 4 KB page, i.e., less than 0.4% memory overhead.

Dual-Dedup provides a convenient API for insertion and lookup operations. Below we present these operations in detail.

Lookup operation

We have designed the lookup operation to be compatible with the page cache. Therefore, it accepts the same parameters as of the page cache lookup function. Now let us look at how Dual-Dedup handles a page cache miss.

Upon encountering a page cache miss, the kernel calls the lookup function. The function receives the address space object of the file and the requested offset within the file as an input. The lookup function then uses this information to calculate the corresponding Logical Block Number (LBN) of the requested data block. Dual-Dedup calculates the LBN using the file-system-specific methods. The result of such a calculation is the first sector of the disk block. Since a disk block is 4KB and a disk sector is 512 bytes, each block consists of 8 sectors. Therefore, we can obtain LBN by dividing the sector by 8.

The lookup operation entails the following steps.

- The calculated LBN is then used to query the Dmdedup's LBN-PBN mapping to get the corresponding Physical Block Number (PBN).
- With this PBN, the lookup function can search its PBN-PCE mapping to get the corresponding page cache entry.

- If Dual-Dedup cannot find the PBN in its mapping, it reports the access as a miss. At this stage, the program control is transferred back to the kernel to proceed with its normal execution.
- However, if the PBN is found in its mapping, Dual-Dedup still needs to ensure that the corresponding page cache entry is not modified from the time it was inserted. For example, consider a scenario where the application reads a file offset and modifies the contents at that offset. During this stage, when the data contents are written to the disk, the PBN corresponding with the offset changes.
- Therefore, Dual-Dedup again calculates the LBN for the page cache entry to determine that its PBN is not changed from the time of insertion in PBN-PCE table.
- At this stage, if Dual-Dedup detects a change in PBN, it removes the PBN-PCE entry.

Insert Operation

When a miss occurs in the page cache, and a miss occurs in Dual-Dedup, the read request must go to the disk. Once the data has been read from the disk, the generic block layer reports read completion to the file system. At this stage, Dual-Dedup needs to insert an entry into its PBN-PCE mapping and ensures that reads with errors do not end up in our mapping and that the offset still exists in the page cache at the time of insertion. The insert operation entails the following steps.

- The address space object and corresponding offset of the page are extracted from the BIO structure.
- The PBN belonging to this PCE (address space object, offset) is calculated in the same way as mentioned in the lookup function.
- Finally, Dual-Dedup inserts an entry in the PBN-PCE mapping, with the PBN as the key and PCE as the value.

CHAPTER 4
EVALUATIONS

We have built a prototype of Dual-Dedup in the Linux kernel, and evaluated its performance with the Ext4 file system as the baseline. In this chapter, we first present the experiment setup and then discuss the results.

Experiment Setup

System Details	
CPU Model	Intel Xeon E5-2683 v4 CPU @ 2.10 GHz
CPU Architecture	64-bit
CPU Cores	32 (2 x 16)
CPU L3 Cache	4 KB
Operating System	64-bit Ubuntu 16.04 LTS
Linux Kernel Version	4.8
File System	EXT4
Storage Device 1 (HDD)	Toshiba MG04ACA100NY (FK1D)
Storage Device 2 (SSD)	Samsung 860 Evo (RVT02B6Q)
Deduplication	
Deduplication Framework	Dmdedup 4.8
Block Size	4 KB
Hash Function	MD5
Metadata Backend	INRAM

Table 1   Experiment setup to evaluate Dual-Dedup

Experiment Results

We use the synthetic I/O benchmarking tool FIO [4] to test Dual-Dedup. FIO tests consists of 3 threads (jobs) each issuing 4KB random reads. We limit the total size of the read to 1GB in reach run and vary the percentage of duplicate across the runs.

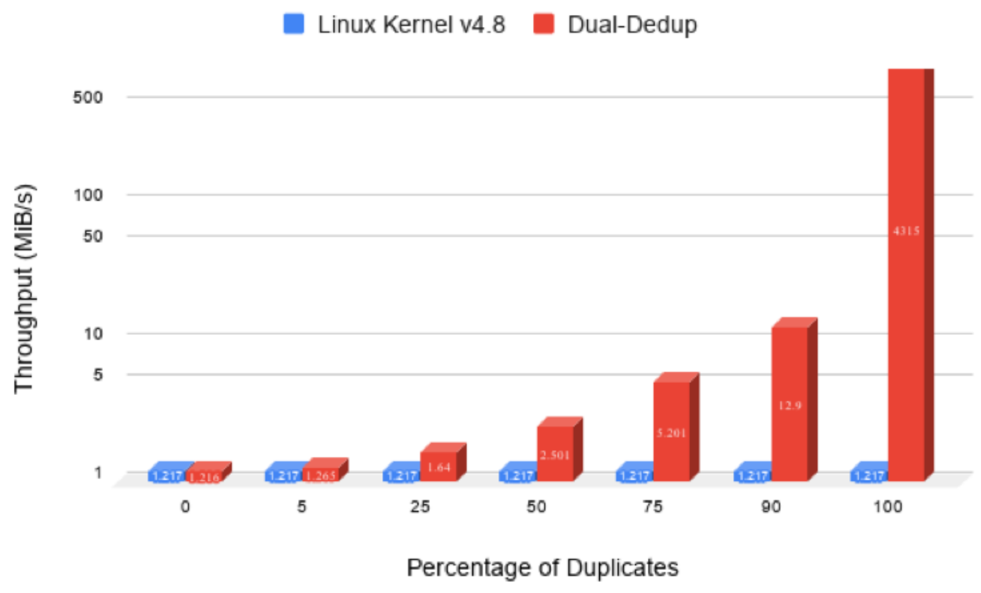

Figure 4 Read throughput on HDD with varying percentage of duplicates among the data for systems with and without Dual-Dedup

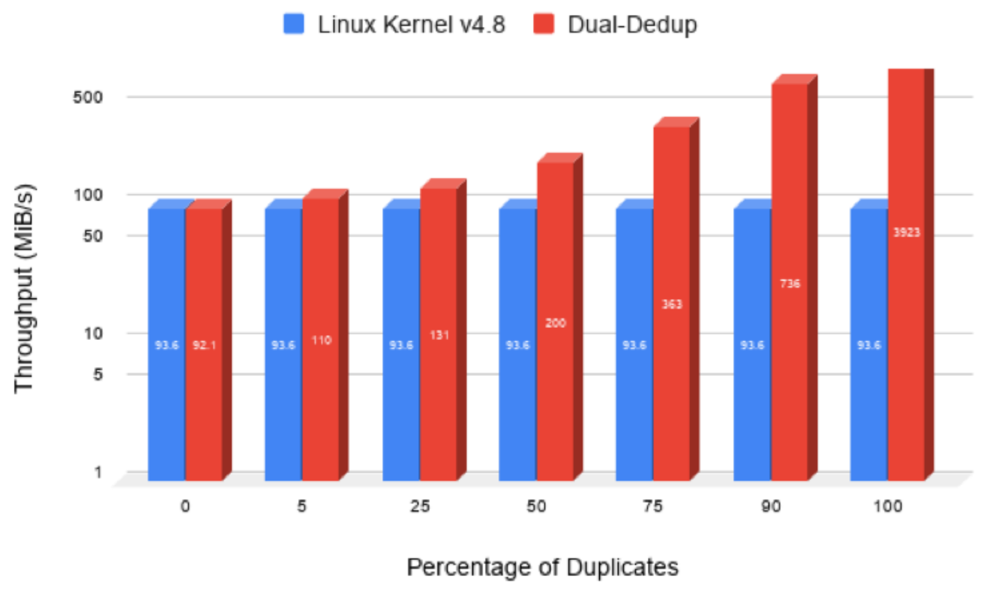

Figure 5 Read throughput on SSD with varying percentage of duplicates among the data for systems with and without Dual-Dedup

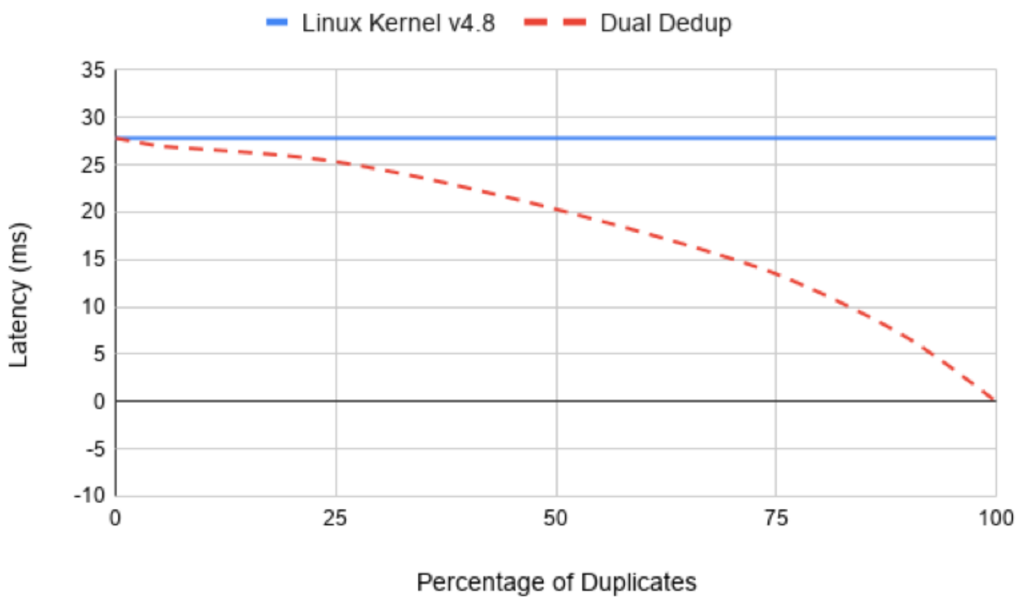

Figure 6 Latency on HDD with varying percentage of duplicates among the data for systems with and without Dual-Dedup

Discussion on Experiment Results

Figure 4 shows the read throughput of systems with and without Dual-Dedup on a HDD. Dual-Dedup shows about 4% improvement in read throughput when there are just 5% duplicates in the data. The more significant improvement can be noted when the data has 25% duplicates, which is a plausible assumption of duplicates in real-world workloads.
Dual-Dedup shows a 34% improvement in read throughput when the data has 25% duplicates. For the stock kernel, the read throughput does not change across the runs, because the kernel does not consider the information about duplicates across the data. Figure 5 shows the results of the same tests on an SSD.

The worst-case scenario for the Dual-Dedup is when there is zero or few duplicates. In this case, Dual-Dedup does not improve performance. As the Figures 4, 5 show, this is indeed the case. Meanwhile, Dual-Dedup does not compromise the performance, which shows that the overhead of our design is extremely low. This overhead is due to the insertions and lookups on the PBN-PCE mapping that we maintain.

Latency is an important metric when measuring data access performance, as it measures how much a process must wait to get the requested data. Figure 6 shows 95th percentile latencies, measured in milliseconds, for the systems with and without Dual-Dedup when the duplicate percentages varied in the accessed data. The stock Linux kernel has high latencies across all the runs because it does not take duplicates into account. However, consistent with the trend of throughput changes, Dual-Dedup has a steady decrease in latency with an increase in the percentage of duplicates in the data. When the data has 25% duplicates, the 95th percentile latency is reduced by 16%, and when there are 90% duplicates, the latency is reduced by 83%.

Implementation of Dual-Dedup does have its space overhead. PBN-PCE mapping is its primary data structure. As mentioned above, each value in this mapping consumes only 16 bytes. Therefore, even in the worst case, the mapping table requires less than 0.4% extra memory for the total amount of unique data on the disk.

CHAPTER 5
CONCLUSION

This thesis presents Dual-Dedup, a scheme to improve the read performance in operating systems by integrating the information about on-disk duplicated with the page cache. It creates a mapping between unique data blocks on the disk and their page cache entries. Dual-Dedup is highly lightweight scheme as it does not detect duplicates by itself. Instead, it efficiently uses the knowledge discovered by the block-level deduplication subsystem. Results from extensive experiments show that Dual-Dedup significantly improves read performance. On FIO tests with 25% duplicate data, our system shows an improvement of 34% in the read throughput when compared with Linux EXT4.

REFERENCES

[1] Tarasov, V., Jain, D., Kuenning, G., Mandal, S., Palanisami, K., Shilane, P., ... & Zadok, E. (2014, July). Dmdedup: Device mapper target for data deduplication. In 2014 Ottawa Linux Symposium.

[2] Koller, R., & Rangaswami, R. (2010). I/O deduplication: Utilizing content similarity to improve I/O performance. ACM Transactions on Storage (TOS), 6(3), 13.

[3] Bovet, D. P., & Cesati, M. (2005). Understanding the Linux Kernel: from I/O ports to process management. " O'Reilly Media, Inc.".

[4] Fio – flexible i/o tester synthetic benchmark. http://git. kernel.dk /?p=fio.git

[5] Meyer, D. T., & Bolosky, W. J. (2012). A study of practical deduplication. ACM Transactions on Storage (TOS), 7(4), 14.

[6] Srinivasan, K., Bisson, T., Goodson, G. R., & Voruganti, K. (2012, February). iDedup: latency-aware, inline data deduplication for primary storage. In Fast (Vol. 12, pp. 1-14).

[7] Yang, Q., Jin, R., & Zhao, M. (2019). Smartdedup: optimizing deduplication for resource-constrained devices. In 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19) (pp. 633-646)

[8] Love, R. (2010). Linux kernel development. Pearson Education.

BIOGRAPHICAL INFORMATION

Ravi Kiran Boggavarapu received his B.Tech. degree in Computer Science and Engineering from Lovely Professional University, India, in 2016. He received his M.S. degree in Computer Science from The University of Texas at Arlington in 2019.