Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Pinto Pedrosa

HIODS: Hybrid Inline and Offline Deduplication System

March 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Pinto Pedrosa

HIODS: Hybrid Inline and Offline Deduplication System

Master Dissertation
Integrated Master in Informatics Engineering

Dissertation Supervised By
João Tiago Medeiros Paulo
José Orlando Pereira

March 2021

ABSTRACT

Deduplication is a technique that allows finding and removing duplicate data at storage systems. With the current exponential growth of digital information, this mechanism is becoming more and more desirable for reducing the infrastructural costs of persisting such data. Therefore, deduplication is now being widely applied to several storage appliances serving applications with different requirements (e.g., archival, backup, primary storage).

However, deduplication requires additional processing logic for each storage request in order to detect and eliminate duplicate content. Traditionally, this processing is done in the I/O critical path (inline), thus introducing a performance penalty on the throughput and latency of requests being served by the storage appliance. An alternative solution is to do this process as a background task, thus outside of the I/O critical path (offline), at the cost of requiring additional storage space as duplicate content is not found and eliminated immediately. However, the choice of what type of strategy to use is typically done manually and does not take into consideration changes in the applications' workloads.

This dissertation proposes HIODS, a hybrid deduplication solution capable of automatically changing between inline and offline deduplication according to the requirements (e.g., desired storage I/O throughput goal) of applications and their dynamic workloads. The goal is to choose the best strategy that fulfills the targeted I/O performance objectives while optimizing deduplication space savings.

Finally, a prototype of HIODS is implemented and evaluated extensively with different storage workloads. Results show that HIODS is able to change its deduplication mode dynamically, according to the storage workload being served, while balancing I/O performance and space savings requirements efficiently.

keywords: deduplication, storage, inline, offline, hybrid

RESUMO

A deduplicação é um técnica que permite encontrar e remover dados duplicados guardados nos sistemas de armazenamento. Com o crescimento exponencial da informação digital que vivemos atualmente, este mecanismo está a tornar-se cada vez mais popular para reduzir os custos das infraestruturas onde esses dados se encontram alojados. De facto, a deduplicação é, hoje em dia, usada numa grande variedade de serviços de armazenamento que servem diferentes aplicações com requisitos particulares (ex.: arquivo, backup, armazenamento primário).

No entanto, a deduplicação adiciona uma camada de processamento extra a cada pedido de armazenamento, de modo a conseguir detetar e eliminar o conteúdo redundante. Tradicionalmente, este processo é realizado durante o caminho crítico do I/O (*inline*), causando perdas de desempenho e aumentos na latência dos pedidos processados. Uma alternativa é alterar o processamento para segundo plano, aliviando assim os custos no caminho crítico do I/O (*offline*). Esta solução requer espaço de armazenamento adicional, visto que os duplicados não são encontrados nem eliminados imediatamente. No entanto, a estratégia a seguir é escolhida de forma manual, não tendo em consideração qualquer possível mudança na carga de trabalho das aplicações.

Esta dissertação propõe assim o HIODS, um sistema de deduplicação híbrido capaz de alterar entre o modo *inline* e *offline* de forma automática considerando os requisitos (ex.: débito do sistema de armazenamento desejado) das aplicações e das suas cargas de trabalho dinâmicas.

Por fim, um protótipo do HIODS é implementado e avaliado exaustivamente. Os resultados mostram que o HIODS é capaz de alterar o modo de deduplicação de forma dinâmica e de acordo com a carga de trabalho, considerando os requisitos de desempenho e a eliminação eficiente dos dados duplicados.

palavras-chave: deduplicação, armazenamento, inline, offline, híbrido

AGRADECIMENTOS

Todo o trabalho desenvolvido durante esta dissertação não seria possível sem o apoio de algumas pessoas fundamentais. Assim sendo, gostaria de tirar um momento para expressar o meu mais sincero obrigado para com essas mesmas pessoas.

Ao meu orientador, Doutor João Tiago Medeiros Paulo, um especial agradecimento por todo o apoio, motivação, assistência e disponibilidade demonstrada durante todo o processo. Também ao meu co-orientador, Professor José Orlando Pereira, por todo o auxílio prestado, principalmente nas decisões mais complexas.

Aos meus colegas do Grupo de Sistemas Distribuídos do HASLab pelo bom ambiente de trabalho proporcionado e também pela assistência prestada.

Por fim, agradeço aos meus amigos e família por todo o apoio e motivação, não só durante a realização desta dissertação, mas sim durante todo o meu percurso académico.

STATEMENT OF INTEGRITY

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

CTA	Context-aware Threshold Adjustment.
DIODE	Dynamic Inline-Offline DEduplication.
DPE	Deferred Priority-based Enforcement.
FF	File Fingerprint.
FLC	Feedback Loop Controller.
HDS	Hybrid Deduplication System.
HPC	High-Performance Computing.
NVME	Non-Volatile Memory Express.
PID	Proportional–Integral–Derivative.
SPDK	Storage Performance Development Kit.
SSD	Solid-State Drive.

1 INTRODUCTION

Nowadays, new products and services are introduced almost daily. These products and services are now generating and consuming digital information at an exponential rate, which was never foreseen to happen so early in this century. An International Data Corporation study [1] suggests that by the year 2025, 73 ZB worth of information will be generated by IoT appliances alone, which will account for 75% of the 56 billion connected devices. A second study by the same corporation [2] also predicts that the total number of data generated in 2025 can reach 175 ZB. Furthermore, a significant share of companies will have at least some part of their business and data hosted in cloud services [3].

In reality, the cloud's many services alongside the highly competitive pricing have led the users to upload a majority of their data to the cloud. Actually, several studies show [4, 5] that the massive cloud storage usage introduced around 50% of duplicate data in primary storage and 90% to 95% in secondary systems. Another study regarding HPC Storage Systems [6] shows averages between 20% to 30% of redundant content, which can rise as high as 70%. The significant amount of duplicates raised the need to eliminate such data, saving storage space and, therefore, costs.

The previous observations led cloud providers to adopt deduplication solutions for their storage infrastructures. Briefly, deduplication is a technique that allows finding and removing duplicate content from a storage system. The deployment of such a solution can optimize the storage space needed by applications since only a single data instance is stored. Then, logical pointers to this single copy are used so that it can be accessed transparently by users storing and accessing the same information.

This technique was initially used in the context of archival and backup systems but it is now being applied to other storage ecosystems such as primary storage, RAM, and SSDs [7]. However, each mentioned ecosystem has a specific set of requirements, which ultimately leads to different solutions since no single system fits them all. For example, secondary storage systems (archival and backup) are only used to store information in the long term. Therefore, in this type of storage, the data is highly unlikely to change.

On the other hand, primary storage is used as the backend by high-performance applications such as databases and analytical platforms. Hence the data stored here is foreseen to be changed by the applications' users. Furthermore, the applications using primary storage may not bear the overheads introduced by traditional deduplication, which calls for an alternative solution.

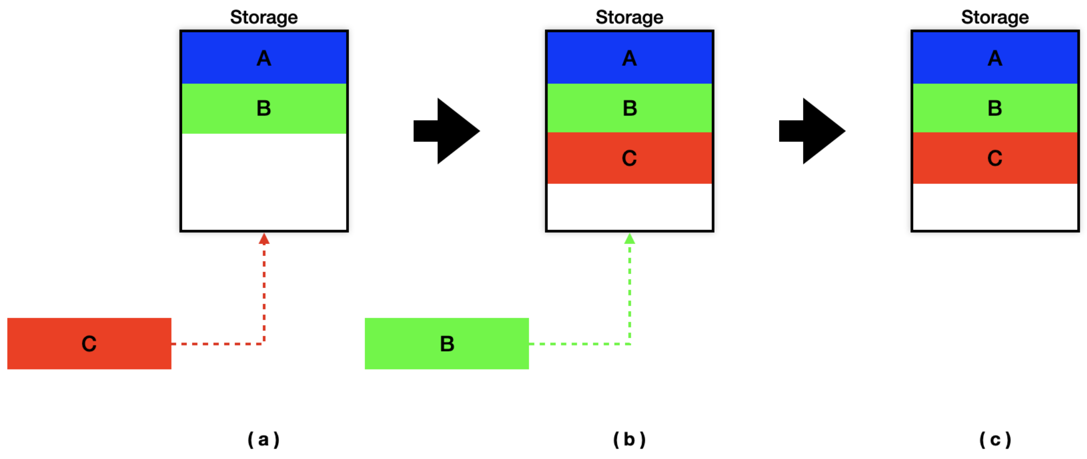

Figure 1: Deduplication Scheme

Traditionally, the deduplication process is able to reduce storage space because it only stores a single instance of the same data. As to quickly identify such duplicate data, all the information already stored on disk must be indexed by the deduplication system. Thus, when a new storage request arrives, the system immediately checks the indexed data for a copy. For example, we can look at Figure 1-ⓐ, where a write request intending to store the data C is pictured. The system then checks the indexed data, which contains A and B, for a copy of the information. As such copy does not exist, C is indexed and stored. In the opposite case, Figure 1-ⓑ, as a copy of B already exists, the data is not written to disk. The final storage organization is seen in Figure 1-ⓒ.

This inline algorithm allows finding and removing redundant data that would, otherwise, be stored on disk, wasting storage space. However, the previously described deduplication process has associated a performance cost since it searches and eliminates duplicate data in the I/O critical path.

Many applications have unique performance requirements, thus may not tolerate the overhead imposed by deduplication processing [7]. Therefore, for this applications to also benefit from deduplication space savings, a new type of deduplication algorithm was proposed in the literature. This new mode, called offline deduplication, differs from the original as it does not find or remove the duplicates in the I/O critical path. Instead, storage write requests are persisted on disk and, later, a background job scans unprocessed information, eliminating it if redundant.

Offline deduplication requires additional storage space since duplicate content is written and only shared later. Also, both applications and the deduplication engine will be accessing concurrently content at the storage medium, thus requiring control access mechanisms to ensure storage consistency and preventing data corruption.

1.1 PROBLEM

In a general fashion, each deduplication system implements the mode that best fits the requirements of applications being served. This process is done statically and manually and, even for applications whose performance requirements may change over time, it is not possible to change the deduplication settings.

Therefore, applications that suddenly contemplate high loads of storage requests and use inline deduplication will experience significant degradation in I/O operations throughput and/or latency. On the other hand, applications working at lower I/O rates, in which inline deduplication could have a minimal impact on I/O performance, may not benefit from additional storage space savings if configured with an offline scheme. However, having a hybrid design that can alternate across both types of deduplication while being aware of workload changes is not a trivial task. Namely, the solution's design must be adapted to integrate both schemes seamlessly, and the applications workloads must be monitored to understand what scheme should be applied at each time.

Additionally, choosing what scheme to use should be done automatically, thus avoiding users or system administrators from making this decision manually.

1.2 GOALS AND CONTRIBUTIONS

Building on section 1.1, this dissertation has as its primary goal the design of an hybrid deduplication system. The proposed solution must be able to dynamically switch between inline and offline deduplication to best fit different application workloads. For example, if an application requires 1000 I/O operations per second and such an objective is not delivered by inline deduplication, the operation mode should automatically be changed to offline, thus reducing the overhead in the critical I/O path and increasing the overall performance.

To achieve the proposed goal, this dissertation's first contribution is HIODS, an *Hybrid Inline and Offline Deduplication System*. HIODS automatically and dynamically ensures that the best deduplication strategy is being applied in order to: i) guarantee the I/O performance goals of different applications; and ii) optimize deduplication space savings.

In more detail, inline deduplication is chosen as the preferred method if the targeted I/O performance objective is being guaranteed, thus also maximizing deduplication space savings. When the I/O workload cannot be supported efficiently by inline deduplication, then HIODS changes the scheme to an offline approach, thus promoting storage performance over space savings. Moreover, HIODS is able to throttle the requests of the offline deduplication engine in cases where concurrent accesses to the storage backend, and the corresponding access control mechanisms, might be affecting the I/O performance of applications. Again, this decision promotes better performance for applications at the cost of extra storage space.

As a second contribution, a prototype of HIODS is implemented using SPDK [8]. SPDK is a framework that provides a wide range of tools and libraries to write high-performance, highly scalable, and user-space storage applications.

Finally, the prototype is extensively evaluated and validated with different workloads, including dynamic ones. The results show that HIODS successfully changed its operation mode based on the system workload and performance objective.

1.3 Dissertation Structure

In the following chapter, we present the state of the art for deduplication systems, while explaining core concepts of this field. Chapter III introduces HIODS general architecture, explains all components, and describes the data flow for supported deduplication modes. In Chapter IV, we discuss the implementation of HIODS prototype. In Chapter V, the testing methodology is detailed and the results from the experimental evaluation are discussed. Finally, Chapter VI concludes the dissertation while pointing interesting future work.

2

BACKGROUND AND STATE OF THE ART

This chapter starts by presenting an overview of the deduplication field and key concepts. Then, we detail relevant related work on the area and discuss its main differences when compared to the solution proposed by this dissertation.

2.1 DEDUPLICATION

The need to reduce the storage space used by archival and backup systems led the industry to search for a mechanism capable of achieving such goals. Thus arises *Deduplication* as a process of finding and removing duplicated data.

Nowadays, deduplication is used in a larger variety of products and services like, for instance, primary storage, RAM, and SSDs [7].

Despite being used with the same end goal and the main elements remain similar between systems, the fact is that the target systems have, quite often, unique characteristics which demand custom made components for deduplication systems to be deployed.

2.1.1 *Basic Architecture*

All deduplication systems aim to reduce the storage space used by applications. To save storage space and reduce the respective costs, the process takes advantage of references and associations, allowing for the elimination of duplicate content without the loss of any information.

In order to find redundant data, deduplication systems must know all the information already stored on disk. Therefore, one of the components widely used in all designs is referred as the **Index**, which indexes all unique data and is instrumental to quickly determine if a piece of information is already in the system. Actually, to improve system performance, a data digest, also called a signature, is often indexed instead of the original data, although the latter can also be used. Furthermore, every entry in the Index is linked with the original data's physical location and its number of references. The former is used to locate the data on disk while the latter tracks the number of logical pointers to the unique copy.

A second fundamental component is called **Metadata**. This data structure associates the address where the application intended to store its data, also known as the logical address (LBA), with its actual physical location (PBA) at the persistent storage medium.

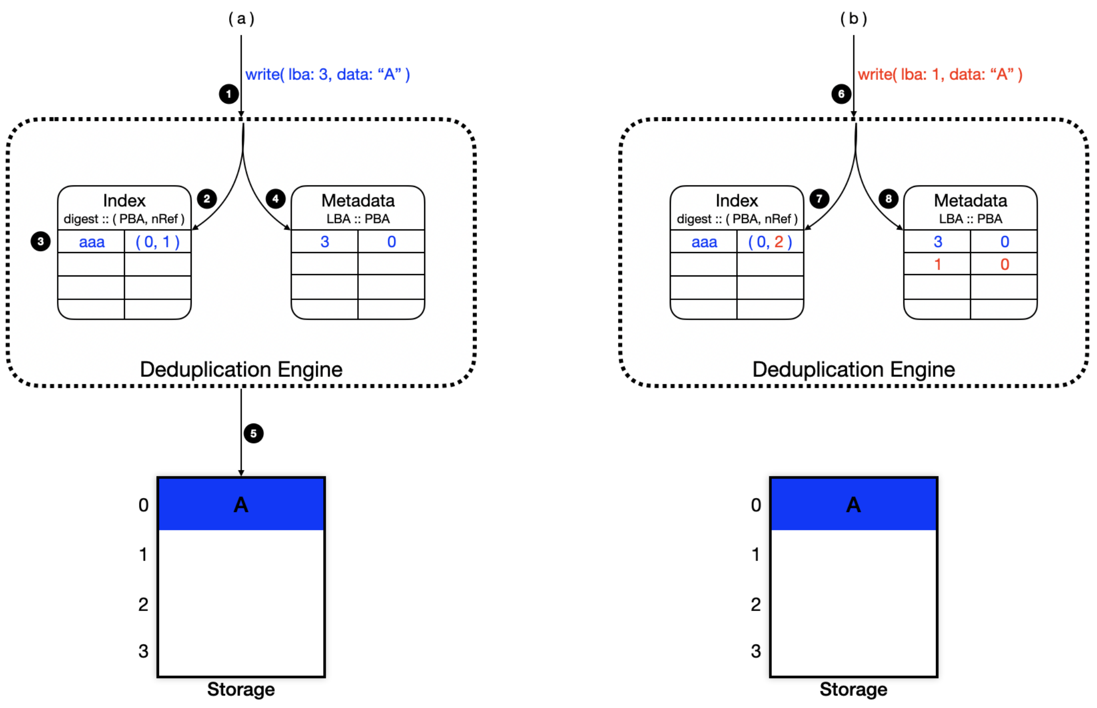

Figure 2: Architecture and Workflow of a Basic Deduplication System

Having shown the essential components, we now introduce the basic workflow of any traditional deduplication system. First, a new storage request containing the data and logical address where it must be stored is intercepted (Figure 2-❶❻). The request is then redirected to the Deduplication Engine, which calculates the data digest (digest(A) = aaa). Next, using such a signature, the Index is able to verify if a copy of the data is already stored (Figure 2-❷). If such data does not exist in the system (Figure 2-ⓐ), an entry that maps to an available physical address is inserted (Figure 2-❸). Then, in Metadata, an association between the request's logical address and the physical block where the data will be stored is created (Figure 2-❹). Finally, the data is written to disk (Figure 2-❺), and the storage request is completed.

Otherwise, if a copy of the data is already indexed (Figure 2-ⓑ), the number of references is incremented (Figure 2-❼), and the associated physical address returned. Finally, the association between the request's logical offset and the returned physical address is created (Figure 2-❽).

Suppose now that the system intercepts a read request to a logical offset. In that situation, the physical address linked with the request's logical offset in Metadata must be read, and its content returned to the application.

2.1.2 *Deduplication Criteria*

A deduplication system can be defined by six fundamental criteria that drive its design and implementation [7].

GRANULARITY

The first criterion and one of the most important is Granularity. It defines the size of the chunks that are going to be used to identify and remove duplicates. Choosing the chunk size is one of the most significant decisions when implementing a deduplication system since it will directly influence its performance and achievable space savings.

When deciding about this criterion, there are three alternatives. The first resorts to whole files (whole file chunking) [9]. This alternative is often used in file-oriented systems since there is no need to split the data into smaller pieces and, therefore, less processing to do in the I/O critical path [10]. On the other hand, two very similar files, where only a single byte differs, will not be identified as duplicates, therefore loosing the opportunity to improve space savings.

The second alternative and a popular one is chunks with a fixed size (ex: 4KB), which is often used in storage systems that already used a fixed-size unit [11, 12]. Unlike the previous alternative, this option can detect redundancy within the file since it compares multiple chunks of the same file. For this very reason, this solution requires extra processing power [10, 13].

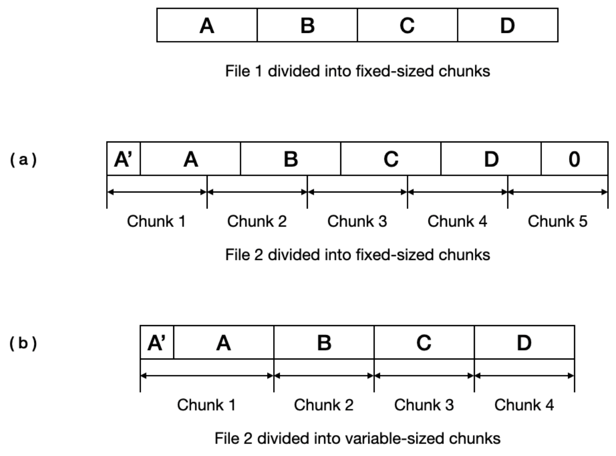
A
B
C
D
File 1 divided into fixed-sized chunks
(a)
A'
A
B
C
D
0
Chunk 1
Chunk 2
Chunk 3
Chunk 4
Chunk 5
File 2 divided into fixed-sized chunks
(b)
A'
A
B
C
D
Chunk 1
Chunk 2
Chunk 3
Chunk 4
File 2 divided into variable-sized chunks

Figure 3: Fixed-Size Chunking versus Variable-Size Chunking

One of the most substantial problems regarding fixed-size chunking is the lack of versatility. For example, let us imagine a base file and a similar second one with an extra 10 bytes added to the beginning. That small 10-byte difference, with the current alternative, will fail to match every single chunk with the original file by 10 bytes (Figure 3-ⓐ). As to mitigate such a problem [14], a variation using variable size chunks [15] was designed. Picking up the previous example, if the original file were deduplicated with a 4 KB chunk size, the modified file's first block could be handled with 4106 bytes. This larger first block would allow the remaining ones to be processed at 4KB granularity, matching the original file blocks (Figure 3-ⓑ).

TIMING

The next criterion is called Timing, and it decides when to perform deduplication. As previously mentioned, traditional inline deduplication [11, 16, 17, 18] detects and removes duplicate data before storing it to disk. Thus, when a storage request is intercepted, a signature that represents its data is immediately calculated. Next, the computed signature is used to check if the request's data is duplicate and, if so, prevent the write operation to disk.

However, this process is performed in the I/O critical path, introducing additional storage performance overhead, which can be critical for some applications [19]. So that high-performance demanding applications can also take advantage of the process, offline deduplication was proposed. In this new mode, all the processing is moved to a background job, which frees the I/O critical path maintaining the original performance. The process of finding and removing redundant content is executed at an opportune moment.

Both options have advantages and disadvantages. The former's main downside is that it introduces a latency overhead in the request since some computations are executed in the I/O path before completion. When it comes to offline deduplication, this drawback is not present, although it has a few of its own. As the data needs to be stored before processing, additional resources are required, including storage and processing power.

INDEXING

The third criterion is called Indexing. It defines the system index, which is the main structure of any deduplication system since it is in it that the digests will be stored.

When it comes to this auxiliary structure, its choice is sometimes overlooked, making it the system's bottleneck. There are three alternatives. The first and most used is called *Full Index*. All distinct data digests are stored with this type, which makes finding all duplicates possible [9, 11]. However, this alternative comes with a high potential for growth, which may become unsustainable over time. In fact, this type of Index has the potential to become so big that its maintainability in memory becomes impossible, triggering the need to access the disk. This change will increase the overhead caused by deduplication since disk accesses are considerably slower than memory ones [11].

An alternative to the previously presented and that solves the size problem is called *Sparse Index* [20, 21]. Unlike the previous option, this type is based on similarities among the chunks. Therefore, the Index does not store signatures representing a single piece of information but referencing a group of similar chunks instead.

Finally, a second alternative that also mitigates the size problem is called *Partial Index*. This option employs the same technique as the Full Index, where each entry represents a unique chunk. However, unlike the first alternative, not all signatures are stored in this solution. In fact, as it can not know all the information already stored on disk, only partial deduplication is achievable [22, 23, 24, 25]. Regarding how to choose which chunks are indexed, there are many algorithms. One of them is the use of access patterns, where the most recent chunks are indexed, and the older signatures are deleted.

LOCALITY

The fourth criterion is Locality, which is not system-related but instead a storage workload property. On the one hand, *Temporal Locality* tells us that a percentage of duplicate blocks are expected to be written in a short time period. On the other hand, *Spatial Locality* explains that if the blocks appeared in a specific order, then they will appear in the same order as before. For example, imagine that the sequence of chunks A, B, C, D, and E enters the system. The spatial locality concept tells us that if a stream formed by A, B, C, and D were intercepted in the future, the next chunk would likely be E.

Accepting this property comes with the need to design and implement additional components to take advantage of it [11]. A component that tries to predict the following intercepted chunk to optimize the index lookups and reduce the overhead is one of many examples of extra complexity in the system. These mechanisms can be quite useful if well designed and implemented and, most of all, when locality is present in the data. However, if the data does not show locality, these mechanisms will not be able to optimize the system, introducing extra overhead. For this reason, many deduplication systems do not take advantage of it [26, 27, 28], a small group focus on only one type, and very few take advantage of both.

TECHNIQUE

The penultimate criterion is called Technique, and it determines how data will be stored on disk, with two main alternatives. The first is the use of unique chunks, storing only unique copies. On the other hand, when dealing with similar blocks, it is possible to store only a base chunk and a list of changes that allow recovering the original data. The former is called *Chunk-based Deduplication* [9, 11], and the latter is known as *Delta Encoding* [29]. When it comes to advantages and disadvantages, the former requires less processing power and can restore faster [30], while the latter allows saving extra storage space [29, 31].

SCOPE

Scope is the sixth and final criterion. It indicates at what level deduplication will be performed in a distributed scenario.

On the one hand, finding and removing duplicates can be done at a *Local Scope* [21, 32], meaning that every node performs deduplication on its own. On the other hand, this process can be performed at a *Global Scope* [12, 26, 28, 33], allowing deduplication to be performed system-wide where shared structures and locking mechanisms are required at the expense of performance.

2.1.3 *Primary Storage vs Secondary Storage*

Deduplication methods are now used in multiple aspects of our lives, particularly in archival/backup and primary storage, among others.

Archival and backup storage share similarities regarding the necessary characteristics needed for a deduplication system since the information present is not expected to change, and both prefer throughput over latency. Furthermore, in secondary storage, a 90-95% degree of duplicate information can be found, causing huge storage losses and increasing the storage costs if such redundant content is stored instead of removed.

There is primary storage on the other side of the coin, which also has unique requirements, but, in this case, regarding storage operations latency. Therefore, the major challenge of a deduplication system aimed at primary storage is to reduce to the maximum the overhead introduced in I/O critical path. Besides latency requirements, in this type of system, data is expected to change regularly. This new update operation, which does not exist in secondary systems, introduces additional complexity to the deduplication process. There is now the need for mechanisms capable of performing the update and simultaneously maintaining consistency with the other referenced chunks. Furthermore, updating information allows for the number of references of individual blocks to reach zero, meaning that its presence in the system must be erased in order to accommodate a new chunk in its physical address bringing storage waste to a minimum. As a side note, primary storage systems are built on top of state of the art hardware, meaning that any reduction in space is directly translated into financial gains.

For all the reasons presented above, the inline mode is often associated with secondary storage and offline deduplication with primary storage. However, many applications require high performance during only a few key moments and could take the overhead of inline deduplication in the remaining occasions. For example, let us imagine an application requiring a high throughput for a few hours every day and that such throughput can only be achieved with offline deduplication. In reality, the application could withstand the overhead introduced by inline deduplication during the remaining hours. However, as the performance for a few hours a day can only be achieved by offline deduplication, a static offline system would probably be implemented. In fact, a hybrid system capable of switching between inline and offline deduplication would be the ideal solution. This way, the system could benefit from inline deduplication during the off-peak hours and switch to offline mode when necessary.

2.2 RELATED WORK

The majority of existing deduplication systems are based on only one timing approach, i.e., they either exploit inline deduplication or perform offline deduplication depending on the workload's performance requirements. Recently, hybrid deduplication systems have emerged as a more efficient solution for workloads whose requirements may vary over time or across different groups of data, and where a combination of offline and inline deduplication may be best suitable.

2.2.1 *DIODE*

A first system that already implements hybrid deduplication is called *Dynamic Inline-Offline Deduplication* (DIODE) [34], and it introduces two new mechanisms. *Context-aware Threshold Adjustment* (CTA) controls inline deduplication, and *Deferred Priority-based Enforcement* (DPE) has the goal to monitor and adjust offline deduplication.

When a file is intercepted by DIODE, it is immediately classified in one of three types having the file extension as the criterion. The first, Highly-deduplicatable Type (H-Type), comprises file extensions known to have a high deduplication gain. Conversely, files belonging to the Poorly-deduplicatable Type (P-Type) have low deduplication gains. All the other files fit in the Unpredictable Type (U-Type).

After this first assessment, the chunks whose original file belonged to the H-Type or U-Type are transferred to DIODE's inline deduplication module, *iDeduper*. However, not all chunks will be processed since it only removes duplicates on redundant chunk sequences higher than CTA's current threshold. This threshold defines the minimum size that a redundant chunk sequence must have to be processed by inline deduplication. For example, suppose that the threshold is set to five. Then, inline deduplication will only be chosen on a five or higher sequence if at least five consecutive requests exhibit duplicate data or if their logical addresses are sequential (ex.: 1, 2, 3, 4, 5). DIODE's choice to only perform inline deduplication to redundant chunk sequences higher than a specific limit allows for improving read efficiency with a slight reduction in redundant data found [19, 35].

P-Type files are skipped and will be dealt with offline deduplication. Therefore, when the number of unprocessed files reaches a threshold now set by DPE, offline deduplication is activated. A sorted by priorities and fixed-size list of undealt files is generated, and the files present will be processed, at a block level, by DIODE's offline deduplication module, *oDeduper*.

Besides the detailed mechanisms above, DIODE also exploits temporal locality since it assigns lower priorities to the most recent accessed files. Furthermore, the full list of written blocks is only processed by deduplication if the system detects more write requests being intercepted. This way, if the system detects multiple read requests during offline processing, the background task is stopped at half the list in order to reduce interference with the read requests.

2.2.2 *D^3*

While the previous system is meant for centralized primary storage, the authors also designed and implemented a distributed version, *Dynamic Dual-Phase Deduplication Framework for Distributed Primary Storage* (D^3) [36]. D^3 shares the core components with DIODE like the extension-based file classification, CTA to control inline deduplication, and DPE to help adjust offline deduplication. However, since it now targets a distributed scenario, D^3 also introduces new components like the *Application Layer*, deployed on the client-side, the *Coordinator Cluster*, and the *Storage Node Cluster* both on the server-side.

In D^3, the dataflow is quite different from DIODE since the files are intercepted on the Application Layer, which is on the client-side. It is also there where they are classified the same way as before (H-Type, P-Type, and U-Type). One of the main differences is the granularity since P-Type files are treated as a whole, and in the remaining types, chunk level deduplication is performed. One other variation is the fact that all file types can be processed in an inline manner.

Once classified, the Application Layer sends the file (P-Type) or the chunks (H-Type and U-Type) to the Coordinator Cluster. In the former, the Coordinator computes its File Fingerprint (FF) and redirects the file to the FF mod n node, where n is the total number of nodes in the system. The node's local inline deduplication module, *iDeduper*, will then search for a duplicate. If such is found, the metadata is updated, and the process is completed. Otherwise, its File Fingerprint is inserted into the File Fingerprint Hash Table, which stores the unique fingerprints for P-Type files only, but the actual process is delayed and performed later by D^3's offline deduplication.

A signature is also computed with the remaining types, and the blocks are sent to the respective node. Like in DIODE, the inline deduplication module will only process redundant chunk sequences higher than the threshold set by the CTA.

All chunks that were not yet processed (P-Type files are later split into chunks) are retrieved by each node's offline deduplication module, *oDeduper*, and sent to the Coordinator

Cluster. When the threshold defined by DPE is reached, global offline deduplication is activated accordingly to the mechanisms already explained in DIODE.

2.2.3 *HPDedup*

Possibly the most significant performance bottleneck in every deduplication system is, without a doubt, the Index. This component, which contains all unique chunks' signatures, tends to become quite large, not allowing its existence in memory. *Hybrid Prioritized Data Deduplication Mechanism* (HPDedup) [37] introduces an *in-memory fingerprint cache* to mitigate such problem.

When a block is intercepted, its digest is computed. After that, the *in-memory fingerprint cache* is queried for the presence of such signature. If present, the physical address is returned, and the pair (logical address, physical address) is inserted into the corresponding structure in the cache. Otherwise, the chunk is written to disk, and offline deduplication will deal with it later.

Besides the in-memory fingerprint cache, HPDedup also introduces the *Stream Locality Estimator*. This module monitors the data arriving at the system to best estimate temporal locality, which is then used to fill the cache to achieve the best hit-rate possible. Spatial locality is also estimated and used to reduce disk read fragmentation.

2.2.4 *Hybrid Deduplication System*

Designing a deduplication system that targets primary storage is not an easy task due to its peculiar characteristics. Because of that, many systems take advantage of data locality in order to improve performance. However, when such systems are cloud-based, locality conditions are not likely to be present due to random accesses. Thus arises *Hybrid Deduplication System* (HDS) [38], which uses similarity to reduce index lookup times.

When a request arrives at the system, it is immediately classified into one of four possible types accordingly to request size and operation: Small Read, Large Read, Small Write, and Large Write. After this initial step, the request is inserted in the corresponding queue and must wait until its queue priority is high enough to be processed.

Afterwards, in the background, the *Request Preprocessing Module* is splitting the write requests' data into chunks and performing additional computations. Among them are data digests calculations.

It is now time to calculate which blocks will be processed and which are not. If it is a small write, all requests will be processed and duplicates are found using a hash table, which maps unique data fingerprints to its metadata. Otherwise, a graph is used to help calculate similarities among chunks. After some processing, the graph is computed, similar blocks are chosen, and metadata is updated.

2.2.5 *Discussion*

To mitigate the challenges that appear when designing a deduplication system that targets primary storage, all systems presented above introduce new algorithms and mechanisms to deal with primary storage unique characteristics.

One of the first mechanisms presented, introduced by DIODE and D^3, is an initial classification based on the file extension. In fact, there is research [39, 40, 41] that associates the extension with the percentage of duplicate data present. Therefore, this approach can be beneficial to the system. However, this mechanism can only be implemented when working at the file-level since it is impossible to determine the original file extension at the block-level.

A second optimization implemented by several of the above systems is only performing inline deduplication on redundant chunk sequences with redundancy higher than a certain threshold. Once again, this approach is also supported by some research [19, 35], which shows that employing this mechanism can significantly improve system performance at extra storage expense.

DIODE and D^3 also introduce some mechanisms to improve offline deduplication. First of all, offline processing is only activated when a certain threshold of unprocessed data backlog is reached. However, since deduplication will be delayed, duplicated blocks will be stored for longer periods increasing the necessary storage space. A second decision implemented by both is prioritizing blocks, which despite introducing extra complexity to the system and requiring more resources, can sometimes be useful.

As a way to reduce the interactions with the Index, since it is almost always a bottleneck for performance [37], HPDedup designs and implements an *in-memory fingerprint cache*. It also introduces the *Stream Locality Estimator* module, which tries to fill the cache to achieve the highest hit-rate possible. If well optimized and working together, this set of mechanisms is an excellent optimization that will undoubtedly bring performance boosts.

The last system presented also introduces a few algorithms. Among them are: splitting I/O into four queues depending on size and operation to perform, different ways to process the write requests, and similarity techniques, which are not very common. Unlike unique signatures, similarity allows the Index to stay compact, which ultimately leads to an increase in performance. Therefore, it makes perfect sense to separate small writes from large writes since similarity is much more likely to exist in large sets.

System	Granularity	Indexing	Locality	Technique	Scope
DIODE	Chunk Level	Full Index	Temporal Locality	Chunk-Based	-
D^3	File & Chunk Level	Full Index	Temporal Locality	Chunk-Based	Both
HPDedup	Chunk Level	Full Index	Both	Chunk-Based	-
HDS	Chunk Level	Sparce Index	None	Chunk-Based	-
HIODS	Chunk Level	Full Index	None	Chunk-Based	-

Table 1: Comparison Among Hybrid Deduplication Systems

The systems previously presented, whose characteristics are compared in Table 1, try to mitigate the overhead introduced by inline deduplication through several mechanisms and algorithms, like locality or caching. For example, when DIODE and D^3 classify a file as Poorly-deduplicatable, which indicates a low duplicate content, it is immediately disregarded as a candidate to inline deduplication. The same happens with HPDedup, which only performs inline deduplication on chunks present in the in-memory fingerprint cache. Finally, HDS only serves read requests immediately, leaving the write ones for a later time. In conclusion, all previous systems employ inline and offline deduplication simultaneously.

HIODS follows a different approach. In this system, only one of the operation modes is active, depending on the workload. Thus, in moments where the application can bear the costs introduced by inline deduplication, that mode is used. Otherwise, the operation mode is dynamically switched to offline deduplication.

A second difference from the previous is when to process the undealt chunks from offline deduplication. For example, DIODE and D3 activate such a process when the threshold defined by the DPE is reached. Despite being able to mitigate the competition for resources, this method requires a higher storage capacity since the undealt chunks will be stored without processing for more extended periods. Therefore, HIODS immediately processes such information to minimize storage waste. Also, if interference with the desired system throughput is detected, a delay between chunks' processing is implemented using Feedback Loop algorithms.

3

ARCHITECTURE

In this chapter, the architecture of HIODS is detailed. Furthermore, the workflow for the different deduplication modes is introduced.

3.1 GENERAL OVERVIEW

All deduplication systems have the same basic architecture, as described in 2.1.1. However, the need for optimizations and the deployment target can lead to modifications in the system's architecture.

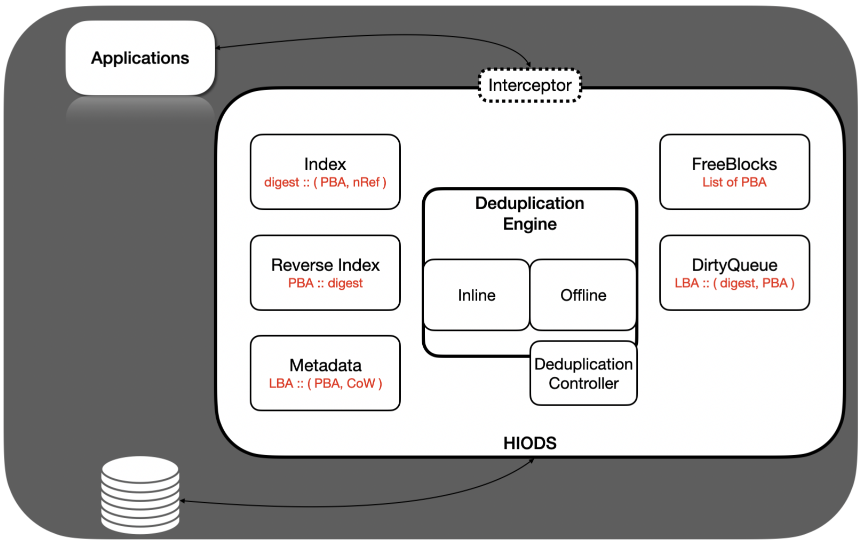

Figure 4: HIODS Architecture

As we can observe in Figure 4, HIODS comprises eight fundamental modules. Some represent auxiliary structures that store and manipulate metadata while others control and perform the deduplication process.

One care from the beginning was to design the system completely independent from its target. Therefore, the **Interceptor** acts as a door to the outside world implementing the most common operations to block devices and storage systems: read(logical address) and write(

data, logical address). The former receives a single argument (logical address) representing the address from where the application wants to read data. It returns the physical address where the data associated with the logical address is actually stored. On the other hand, the latter receives as arguments the data to store (data) and the address where to store it (logical address). As a response, this function returns an available physical address, if available, where the data must be stored instead. Otherwise, an error code (-1) is returned.

Besides being the gateway to the system, this module also collects metrics, like throughput, latency and entry/exit timestamps, that will help dynamically adjust the deduplication operation mode.

The main module where duplicate data will be found and eliminated is called **Deduplication Engine**. Depending on the operation mode currently set, this component either will find and remove redundant data in the I/O critical path, or later perform offline deduplication. The module responsible for adjusting the deduplication mode is the **Deduplication Controller**. With the metrics sent by the Interceptor, this module is able to calculate the system load and decide if the deduplication mode should be switched or if such is not necessary.

The remaining modules provide structures and manipulation functions to assist the Deduplication Engine. As to find and remove all redundant data being stored, the deduplication system must know exactly which data it already has stored on the disk. Therefore, the **Index** stores a signature, also known as digest, for each unique chunk. Furthermore, every signature is also linked with the corresponding information's physical location (PBA) and the number of references using such data (nRef). The former allows for quickly finding duplicate data, and the latter indicates the number of references to a physical address (*i.e.*, logical pointers to the unique chunk). When the number of references reaches 0, then the physical address is no longer being used and can be allocated to store another unique chunk.

The **Reverse Index** was developed in order to provide an efficient mechanism to search for the hash of a given chunk when only knowing the chunk's physical location at the storage medium. This is important when shared chunks are updated, and the number of references needs to be updated.

As previously mentioned, one of the main concerns in a deduplication system is to link the logical address where the application intended to write in the first place with the actual physical address where the data was written. Such a task falls under the **Metadata** component, which links a logical address (LBA) to a physical address. Furthermore, every entry also has a Copy-on-Write (CoW) flag indicating if such data is already being shared among multiple logical addresses. The CoW flag is important because stored chunks cannot be updated until there are no logical addresses referencing them.

HIODS final components are the **FreeBlocks** and the **DirtyQueue**. The former contains a list of available disk addresses where novel chunks can be written. Finally, so that deduplication can occur later, the DirtyQueue stores, for each undealt storage request, the data digest computed from the original data, the logical block address present in the original request, and the physical block address where the data was physically stored.

3.2 DEDUPLICATION WORKFLOW - INLINE MODE

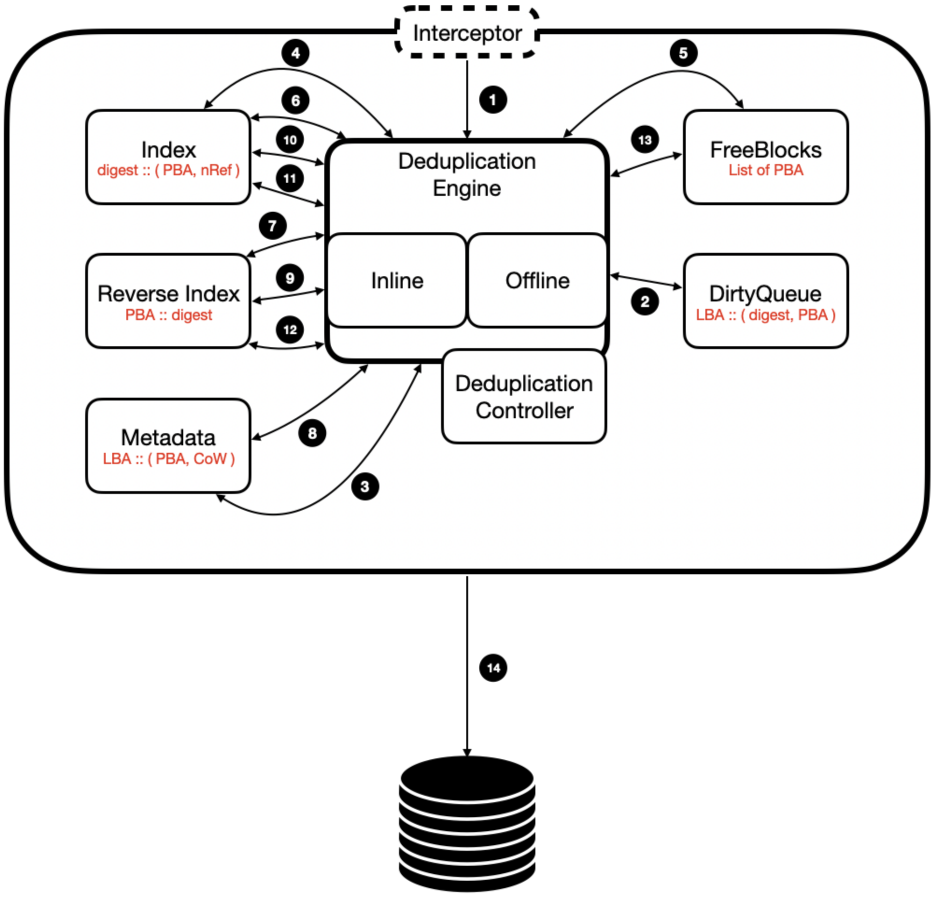

Figure 5: Deduplication Workflow on Inline Mode

When a new storage request arrives at the Interceptor, an entry timestamp is computed, and the request is redirected to the Deduplication Engine for processing (Figure 5-❶). Assuming inline deduplication is active, the first step is to compute a digest of the request's data.

Next, and since HIODS is a hybrid system where the operation mode changes between inline and offline, the DirtyQueue is checked (Figure 5-❷) for a pending request for the same logical address, i.e., a request that was processed by offline mode but was not yet deduplicated. If such exists, it can be removed from the DirtyQueue, because the new storage request makes the pending one obsolete as it rewrites the previous information. At last, the physical address where the previous data was written, and present in the DirtyQueue, is returned to FreeBlocks.

After the previous verification, the Metadata module is consulted (Figure 5-❸) to retrieve information associated with the storage request's logical address. This query returns the associated physical address and the respective Copy-on-Write flag, which indicates if the returned address is being shared among multiple logical addresses. If the CoW flag is set, additional steps are required to ensure the deduplication system's consistency.

Let us assume that other logical addresses do not share the returned physical address (CoW == 0). It is now necessary to determine if a copy of the request's data is already stored on disk. As to ascertain such, a *test_and_increment* operation is conducted on the Index (Figure 5-❹) with the previously computed digest as key. This operation allows for determining if the digest is already in the Index, indicating whether the data is duplicated. If this is not the case, a new physical address is requested from FreeBlocks (new_phy_addr) (Figure 5-❺). Next, the pair (digest, new_phy_addr) is inserted into the Index (Figure 5-❻) and the reverse pair into the Reverse Index (Figure 5-❼). Finally, Metadata is updated (Figure 5-❽) with the logical address associated with the request now pointing to the new physical address. Also, the CoW flag is set, indicating that, from now on, the beforementioned physical address is being shared. On the other hand, if the digest already exists in the Index, the *test_and_increment* operation previously performed increments the associated number of references and returns the physical address where the data is stored (phy_addr). In this case, the last step is to point the logical offset in Metadata (Figure 5-❽) to the physical address where the copy of the request's data is stored (phy_addr), also activating the CoW flag.

In the case where the data is already being shared (CoW == 1), the same operations are performed. However, it is also necessary to check if the data there stored is still referenced by any other logical address (nRef > 0). With that in mind, the Reverse Index is consulted (Figure 5-❾), with the physical address returned by the Metadata module as the key, retrieving the digest of the data there stored. Next, its number of references is decremented and returned by the Index (Figure 5-❿). Suppose the returned value equals zero, indicating that the data is no longer being used by any logical address. In that case, the digest is removed from the Index (Figure 5-⓫), and the physical address returned by Metadata is removed from Reverse Index (Figure 5-⓬). This phase's final step is to return to Freeblocks (Figure 5-⓭) the same physical address removed from Reverse Index.

Finally, if the data has to be written to disk, the physical address provided by Freeblocks is returned to the Interceptor. This component then calculates the timestamp and sends the previous timestamp along with the recently calculated one to the Deduplication Controller. At last, the data is written to the new_phy_addr address on disk (Figure 5-⓮), and the write request is marked as completed.

3.3 DEDUPLICATION WORKFLOW - OFFLINE MODE

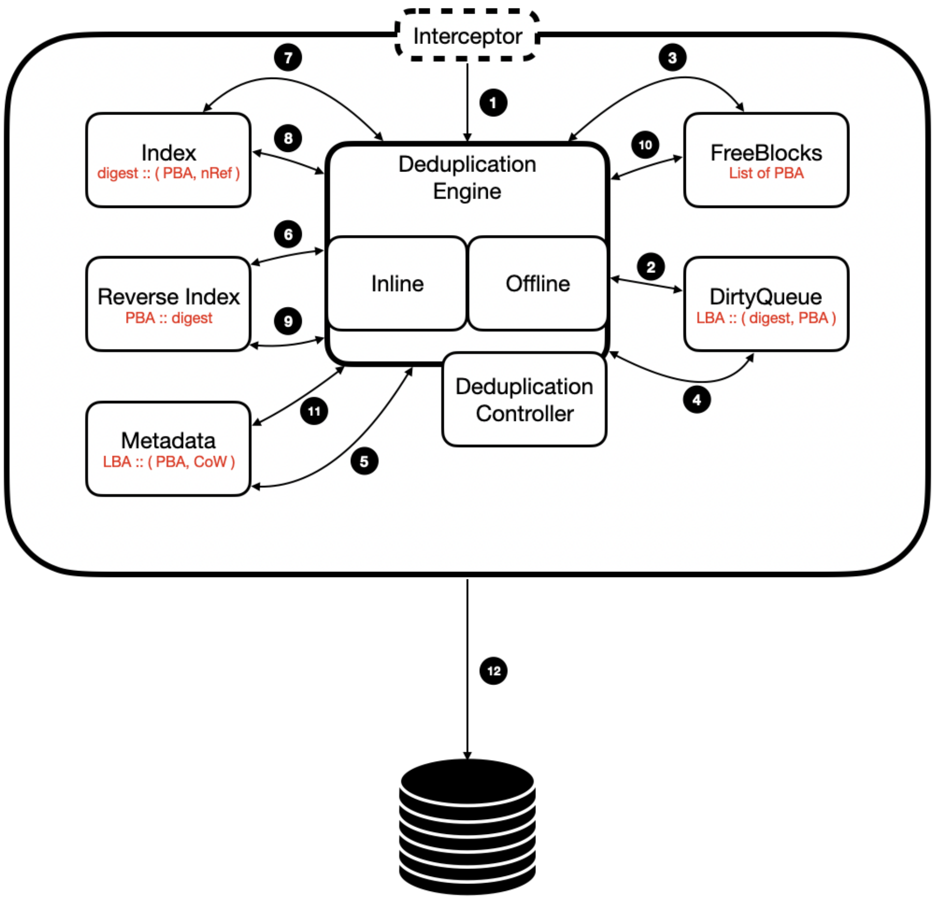

Figure 6: Deduplication Workflow on Offline Mode

The beginning of the process is quite similar to the one described above. When the Interceptor receives a storage request to process, the timestamp is collected, and the request is redirected to the Deduplication Engine (Figure 6-❶).

Next, a pending deduplication request to the same logical address is searched for in the DirtyQueue (Figure 6-❷). In fact, if there was a request to the same logical address that was not yet processed, it became obsolete due to the current rewrite. Therefore, we can use the same physical address to store the data present in the current storage request. If an old request did not exist, a new physical address is requested from FreeBlocks (Figure 6-❸).

So that deduplication can be performed later, it is necessary to store some information to make it possible. Therefore, to avoid disk operations and increase system efficiency, a data digest is calculated and stored in the DirtyQueue (Figure 6-❹), alongside the original logical address and the previously chosen physical address.

Following the previous step and before updating the Metadata, it is necessary to check whether the physical address currently associated with the logical offset in Metadata is being shared among multiple logical addresses. As to verify such condition, the Metadata module is consulted (Figure 6-❺) with the request's logical address as the key returning the associated physical address and the corresponding CoW flag. Suppose the address is being shared (CoW == 1). Then the digest associated with the physical address is returned by Reverse Index (Figure 6-❻), and its number of references is decremented in the Index (Figure 6-❼). If the number equals zero, the digest is removed from the Index (Figure 6-❽), and the pair (physical address, digest) deleted from Reverse Index (Figure 6-❾). To end the current phase, the physical address is returned to FreeBlocks (Figure 6-❿).

Finally, an update to Metadata (Figure 6-⓫) makes the request's logical address point to the physical address chosen in step 2 or 3. As deduplication has yet to be performed, the CoW flag is set to zero. The physical address is returned to the Interceptor, which calculates the current timestamp and sends it alongside the initially calculated one to the Deduplication Controller. At last, the data is written to the physical address returned to the Interceptor (Figure 6-⓬).

3.4 DEDUPLICATION WORKFLOW - BACKGROUND PROCESSING

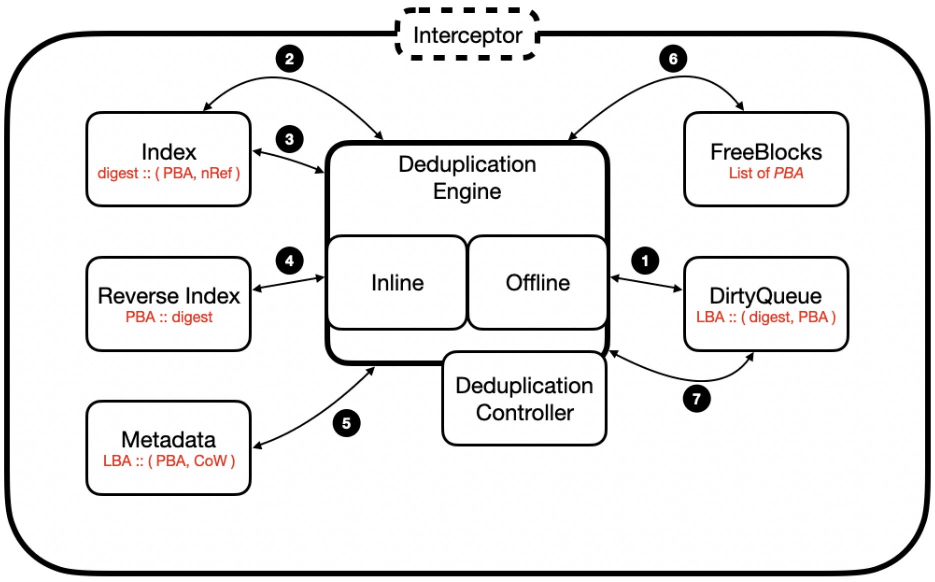

Figure 7: Background Process Workflow

As offline deduplication only stores the necessary information for later processing, not finding nor removing duplicates in the I/O critical path, a background task is required to perform the process.

Therefore, this process, which is only activated periodically, for example, 1 minute after the last time it went to sleep, has the primary goal to process the dirty blocks, i.e., the entries in the DirtyQueue. Each dirty block comprises a data digest and a logical address, both present in the original storage request and the physical address (new_phy_addr) where it was stored on the disk by the foreground I/O logic.

The following algorithm is followed for every entry in the DirtyQueue. First, a dirty block is retrieved from the DirtyQueue (Figure 7-❶). Next, a *test_and_increment* operation is performed on the Index (Figure 7-❷) using the digest present in the dirty block as the key. This operation allows for determining if the digest is already in the Index, indicating whether the data is duplicated.

If the previous operation replies that the digest is not present in the Index, and since the data is already stored on the disk, it is only necessary to insert the digest and the physical address where the original data was previously stored (new_phy_addr) in the Index (Figure 7-❸). The reverse pair is inserted into the Reverse Index (Figure 7-❹). The final step is to update the Metadata (Figure 7-❺) pointing the original logical address to the physical address. The CoW flag is also set, indicating that multiple logical addresses could be using that same physical one from now on.

Otherwise, i.e., if the digest is already present in the Index, the number of references associated with it is incremented, and the corresponding physical address is returned (phy_addr). Next, it is necessary to point the logical address present in the dirty block to the physical address returned by the Index in Metadata (Figure 7-❻). As to finish this step, the physical address present in the dirty block (new_phy_addr) must be returned to FreeBlocks (Figure 7-❻) since both phy_addr and new_phy_addr store the same data on disk. This step is required to optimize the used space, eliminating copies of duplicate data on disk, which is the primary goal after all. Finally, the DirtyQueue entry is expunged (Figure 7-❼).

Being a background job, it competes for resources with the primary process. Such competition may cause concurrency issues as both jobs may access the same structures for processing. In order to mitigate that problem, concurrency control mechanisms for all auxiliary structures were also implemented.

3.5 DEDUPLICATION CONTROLLER

In a vast set of systems, deduplication is performed in a static manner, i.e., only one mode is implemented. However, HIODS primary goal is to update the operation mode dynamically, inline to offline and vice-versa. The exact moment when the mode should switch is one of the most significant challenges in designing an hybrid deduplication system.

In order to achieve such goal, HIODS introduces a new way to switch the operation mode automatically. Namely, it relies on two conditions to switch its operation mode. The first is the system load, calculated by the Deduplication Controller, and the second is the system performance. Therefore, when the application starts to send a high number of storage requests, the Controller is able to realize that the Engine is under considerable stress and may not be able to achieve its objective. After this realization, the current throughput is acquired and compared with the desired performance. Suppose the former is lower than the latter. In that case, the operation mode is switched to offline and kept that way until the Deduplication Controller decides that the system is no longer under heavy load, in which case, the operation mode is switch back to inline deduplication.

As it is possible to observe by the workflow images from the previous chapters (3.2, 3.3), inline deduplication does more operations in the I/O critical path than offline deduplication. Such is due to the fact that the latter only finds and removes duplicates later in time and on background.

As to find and remove the redundant data that offline deduplication misses, a background job is required. However, when to activate the process does not have a trivial answer. Actually, when activated regularly, an additional amount of resources, which may be necessary to the primary job, will be used by the background process. On the other hand, if the background process is only activate in periods of low I/O load, it will not be able to find and remove duplicates quickly, thus wasting valuable storage space.

In HIODS, the DirtyQueue is processed periodically to eliminate as many duplicates as possible, as soon as possible. However, the need to share resources previously mentioned is contemplated. Namely, our solution implements a *Feedback Loop Mechanism* in order to control the throughput of background deduplication processing. If the target performance objective is not being met, this mechanism calculates a time, in nanoseconds, to delay the processing of the entries in the DirtyQueue. This pause in the background job will allow the foreground process to increase the throughput and achieve the desired performance target.

Feedback Loop Controllers, also known as *Closed Loop Controllers*, are a type of controller that receives input measurements from the same system where they are implemented. In this particular case, it receives the current throughput and calculates the time, in nanoseconds, to delay the background process in order to achieve the target throughput.

4 PROTOTYPE

To be able to assess the mechanisms introduced in the previous chapter, a prototype of the system was developed. To achieve that, C was chosen as the programming language and SPDK as the framework where HIODS was integrated.

4.1 SPDK

Every system's performance is always limited by the weakest link, and storage systems are no exception. Until a few years ago, these systems were limited by hardware. However, in the last few years, the hardware business has changed considerably with the appearance of SSDs and, more recently, with NVMe SSDs. All these innovations caused the balance to change sides, and, nowadays, it is the software that introduces overhead to the system.

Considering what was previously mentioned and according to tests [8], the Kernel I/O Stack causes a significant percentage of the software overhead due to context switches, data copy between the kernel and userspace, interrupts, and resources competition. Thus arises SPDK, short for Storage Performance Development Kit, with the primary goal to bridge the gaps previously presented, which means reducing the overhead introduced by software, which ultimately translates into high-performance applications. SPDK achieves its goals with three techniques [8]:

- Move all necessary drivers into userspace, which avoids system calls and enables zero-copy access from the applications;
- Polling hardware for completions instead of relying on interrupts, which lowers both total latency and latency variance;
- Avoiding all locks in the I/O path, instead relying on message passing.

Besides being built with performance as the primary focus introducing all the above mechanisms to improve it, SPDK was also designed to be modular and extensible. Thus, this framework introduces a layered structure based on block devices that can be stack on each other.

SPDK presents two types of block device modules. The first is called a block device, and its differentiating characteristic is that it represents physical devices, i.e., devices that can store information. On the other hand, there are virtual block device modules. These receive storage requests, process them in some manner, like deduplication or encryption services, and finally, redirect them to the underlying block device, either physical or virtual.

HIODS was developed as a virtual block device module. It receives requests from applications, finds and removes redundant data if inline deduplication, or makes the necessary arrangements for later processing if the offline mode is activated. Finally, if necessary, redirects the request to the NVMe block device for persistent storage.

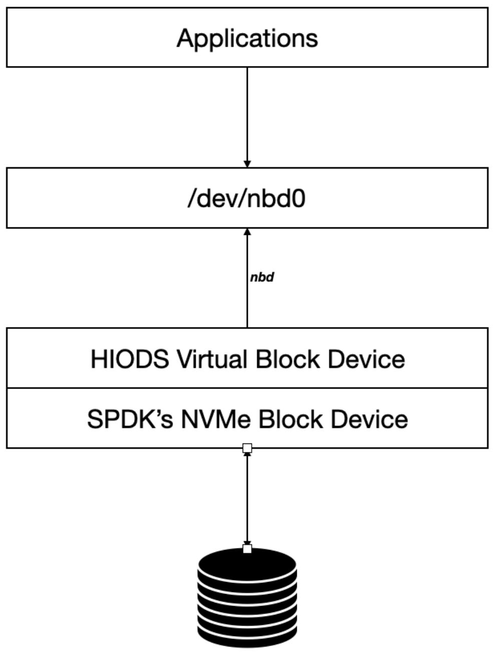

Figure 8: HIODS integration with SPDK

So that applications can take advantage of HIODS virtual block device, such is exported as a regular operating system block device by Network Block Device (NBD) [42], as shown in Figure 8. If this were not the case, the applications would need to be changed in order to integrate with SPDK.

4.2 IMPLEMENTATION DETAILS

With the extra deduplication processing since I/O requests are intercepted until their completion, a decrease in system performance is expected. In order to accurately measure such performance drop, it is necessary to implement the most capable system possible. Thus, the first version of the deduplication system had all its data structures stored in memory.

Therefore, the Index was implemented as a *GHashTable* from *glib* [43] with the data digest as the key. Reverse Index used an array where each position represented a physical address, with its content being the digest of the data there stored. In the same way, Metadata also implemented an array where each position corresponded to a logical address, and its content was the associated physical address and CoW flag. Freeblocks was stored using a circular array where all available physical addresses were stored. This component's solution also involved an index for retrieving the associated physical address and a second one for storing a returned address. The final structure, DirtyQueue, once again resorted to an array where each position matches a logical address. Every position stores the corresponding digest and physical address where the data represented by the digest was stored by offline deduplication.

However, a system where its data is stored in memory does not allow for its long-term maintenance and scalability. Therefore, HIODS' second version was developed with the primary goal to promote persistent storage of metadata.

Considering the data types store by every component, is it perceptible that all of them associate key-value pairs. For instance, a data digest is associated with the physical address and the number of references in the Index. Therefore, all the previous components could easily be stored in Key-Value Databases. After searching for possible candidates, leveldb [44] was chosen to store the Index, Reverse Index, and Metadata. Actually, the DirtyQueue remained in memory as it is not a vital component. While the loss of the data stored in the three first mentioned components would lead to the system's inconsistency, the loss of DirtyQueue would only lead to some lost deduplication opportunities.

Finally, the last component with the need for persistent storage was FreeBlocks. In reality, as it only stores a list of available physical addresses, the first implementation resorted to a file. However, it quickly became apparent that such implementation introduced significant overheads. Therefore, the solution was to store this structure in a dedicated disk partition accessible through SPDK. Nevertheless, if every single request/return of an available physical address required accessing the disk, it could lead to unwanted pressure on this vital component. With that in mind, a configurable caching mechanism, currently set for 16000 blocks, was designed and implemented.

4.3 DEDUPLICATION CONTROLLER

HIODS' primary objective is being able to switch between deduplication modes whenever necessary. To achieve such a goal, the Deduplication Controller was developed.

As previously mentioned, the switch between inline and offline deduplication is based on two conditions: the current system load and the desired performance goal. So that the former can be calculated, the system uses the entry time of the current request and the exit timestamp of the previous request. Actually, we assumed that when a system is under heavy workloads, a new request is dealt immediately after the previous one exits the system. Therefore, if the difference between the previous two timestamps is under a certain limit, 3 ms in our implementation, the current request is flagged as "in stress".

Periodically, a background job is responsible for determining if the system is under heavy stress. To label it as such, the ratio between the total number of requests and the ones marked as "in stress" must be at least 90%. Furthermore, the last request must have been caught within a few moments before the evaluation, 1 second in our implementation, to eliminate unique bursts as false indicators. For example, let us imagine that only ten requests in a row were intercepted right after an evaluation. As they were made in a row, the first condition is verified. However, it was a solo burst that should not trigger the Controller to detect stress. Therefore, the second condition helps in avoiding such situations. If both conditions are verified, the Controller realizes that the Engine is under stress and may not achieve the desired performance goal.

Suppose the first condition is verified. Then, the current system throughput is acquired and compared with the desired performance. Suppose now that the first condition is verified and the target performance is not reached for a consecutive number of measurements in a row, 3 in our implementation. In that case, offline deduplication mode is activated, easing the I/O critical path's processing, increasing the overall system throughput. For inline deduplication to be switched back, the only requirement is for the Engine not to be under heavy workloads for a certain number of measurements in a row, once again, 3 in our implementation.

4.3.1 *Feedback Loop Controller*

In order to control the background job's throughput, we endowed the deduplication system with a *Feedback Loop Controller* that delays the processing of dirty blocks. This mechanism controls the number of dirty blocks processed every second, decreasing the interference between foreground and background I/O operations, thus helping the foreground job to reach and maintain the target performance.

When it comes to this category, there were several possible choices. One of the most known and used in the industry is the *Proportional–Integral–Derivative Controller* [45]. This algorithm was chosen to integrate this system.

```
integral = 0
previousError = 0
def pidController( measuredThroughput , dt ):
    delay = 0
    error = 0
    derivative = 0
    kp = 5.0 , ki = 0.15 , kd = 0.25

    error = targetThroughput - measuredThroughput;
    integral = integral + error * dt;
    derivative = ( error - previousError ) / dt;

    delay = kp * error + ki * integral + kd * derivative;
    previousError = error;

    return delay;
```

Listing 4.1: Feedback Loop Controller

One of the characteristics of this type of Controller is the fact that it receives as input a measurement from the system where it will be deployed. Listing 4.1 presents the pseudo-code for the PID Controller implemented in HIODS.

As we can observe from the pseudo-code, this PID algorithm receives as an argument the system performance (*measuredThroughput*) measured over a period of time (*dt* in seconds). From the former, it is possible to calculate the error, i.e., the Proportional component, by subtracting the measured to the desired throughput (*targetThroughput*). Next, the remaining two components, Integral and Derivative, are computed. Afterward, each component is multiplied by its coefficient, k_p, k_i, and k_d. Finally, the time to delay the processing of dirty blocks, in nanoseconds, is obtained by adding the previously multiplied components.

5 EXPERIMENTAL EVALUATION

To ensure the correct implementation and evaluate the mechanisms introduced, as well as assess the overall performance of the deduplication system, a series of tests were designed and executed.

At an early stage, a preliminary set of experiments with the primary goal to evaluate the overhead introduced by SPDK was developed. The next collection, which we called *micro experiments*, allowed for evaluating critical operations individually. This set also helped in adjusting specific system parameters. Finally, with the *macro experiments*, it was possible to observe and evaluate all mechanisms working together.

5.1 TESTING METHODOLOGY

In order to perform the experiments, a tool capable of submitting I/O operations to a block device was required. DEDISbench [46] was chosen for this purpose.

This tool is capable of performing read and write operations in three different modes: sequential, uniform, and hotspot. The first reads/writes to sequential disk addresses, and the second chooses the target address according to an uniform distribution. Finally, the hotspot mode focuses most of its operations on a small set of disk addresses.

Furthermore, DEDISbench also allows for configuring the target, i.e., where to write/read data, the amount of information to write/read (operation size), the block size, i.e., the unit to which the block device works, and the number of processes to use.

DEDISbench distinctive characteristic is that it takes as argument a distribution file that specifies a realistic content distribution, extracted from analyzing real datasets. This distribution is then used to generate realistic content for write requests at the targeted system under evaluation. For these specific tests, two distributions were primarily used: highperf and kernels. The former contains a small percentage of duplicate data (around 20%) and the latter presents around 75% of redundant information.

To proceed with the designed tests, both DEDISbench and SPDK were properly installed in a single machine. This machine was equipped with one processor (Intel(R) Core(TM) i5-9500 CPU @3.00GHz), which contained six cores (1 thread per core), 16GB of RAM, an

500GB hard drive (WDC WD5000AZLX-75K2TA1 500GB) with Ubuntu 18.04 installed, and a 250GB NVMe SSD (Samsung SSD 970 EVO Plus 250GB).

In general, the performed experiments focused on read and write operations in sequential, uniform, and hotspot modes, and used one and four processes. Each test had 20 minutes or 64GB duration and was run three times to obtain the mean and standard deviation. Finally, it is worth noting that hardware metrics (CPU, RAM, and disk) were retrieved using dstat [47] in every test across all categories.

5.2 PRELIMINARY EXPERIMENTS

This initial test set had the primary goal of observing the performance losses that arrive from using SPDK. In order to achieve such an objective, intensive I/O tests were performed. Firstly, NVMe was directly targeted. With this mode, the best possible performance was achieved since there were no additional layers. Later, the tests focused on the same disk exported as a network block device by SPDK. This initial round consisted primarily of read and write requests under two modes: sequential and random. For these experiments, DEDISbench was configured to generate the highperf content distribution.

5.2.1 *Results Analysis*

Target	Mode	Throughput (MB/s)		Latency (us)	
		Mean	SDeviation	Mean	SDeviation
NVMe	Sequential	1724,55	1,89	2,00	0,00
	Uniform	46,16	0,01	84,00	0,00
SPDK	Sequential	1547,46	2,33	2,00	0,00
	Uniform	48,66	0,04	80,00	0,00

Table 2: Read Operations on NVMe vs SPDK

Focusing on Table 2 regarding read requests, a substantial decrease in performance between the sequential and random modes is immediately noticeable (NVMe: 1724 MB/s to 46 MB/s, SPDK: 1547 MB/s to 49 MB/s). Those were expected since random read operations are more costly to a disk than sequential ones.

Moving on to compare the distinct environments in the same category, it is evident that, from NVMe to SPDK, the throughput on sequential reads decreased, from 1724,55 MB/s on NVMe to 1547,46 MB/s on SPDK. Such a situation was entirely expected since the number of intermediate layers has increased. Besides SPDK itself, the use of nbd to export the disk also causes additional overhead.

On the other hand, the uniform reads performance increased slightly, from 46,16 MB/s on NVMe to 48,66 MB/s on SPDK. Since this type of I/O is conditioned by hardware and not software, a small variation was expected.

Target	Mode	Throughput (MB/s)		Latency (us)	
		Mean	SDeviation	Mean	SDeviation
NVMe	Sequential	504,87	0,48	6,00	0,00
	Uniform	529,03	0,85	6,00	0,00
SPDK	Sequential	462,24	2,28	7,00	0,00
	Uniform	445,12	1,33	7,00	0,00

Table 3: Write Operations on NVMe vs SPDK

Concentrating on the writes, whose results are presented in Table 3, we can quickly identify a contrast between sequential and uniform modes.

Actually, NVMe results show an increase between sequential and uniform modes (504,87 MB/s on sequential and 529,03 MB/s on uniform mode), but SPDK results present a decrease between the two (462,24 MB/s on sequential to 445,12 MB/s on uniform mode). In fact, uniform write operations in SSDs, unlike in HDDs, are almost as efficient as sequential ones, which may lead to negligible differences between the two modes.

Regarding the results between the two environments, sequential performance drops from 504,87 MB/s to 462,24 MB/s and from 529,03 MB/s to 445,12 MB/s on uniform mode. Once again, such a decrease is caused by the additional layers introduced with SPDK.

Target	Operation	Mode	CPU (%)	Memory (GB)
NVMe	Read	Sequential	2,40	0,64
		Uniform	0,04	0,64
	Write	Sequential	3,38	0,69
		Uniform	2,95	0,65
SPDK	Read	Sequential	11,08	6,83
		Uniform	8,94	6,62
	Write	Sequential	9,59	4,01
		Uniform	9,98	4,14

Table 4: Resources Used by NVMe and SPDK

It is possible to observe from Table 4, which compares the resources used by both tests, that the experiments involving SPDK used around 10% of the available processing power. On the other hand, NVMe tests required significantly less power (around 3%). Furthermore, the tests using SPDK required more memory than on NVMe (an average of 5 GB versus less than 1 GB with NVMe). Such a discrepancy was expected since SPDK introduces new processing layers, which require additional resources.

5.3 MICRO EXPERIMENTS

This second category had the primary objectives of guaranteeing the system's correct implementation and evaluating each component independently. Furthermore, it also allowed observing the overhead introduced by deduplication.

As already detailed in chapter 3, the hybrid deduplication system presented in this dissertation comprises four main components: inline mode, offline mode, the mechanism for switching between the previous, and the Feedback Loop Controller. This series of tests enabled the analysis of the first and second components, as well as to adjust the last one's parameters.

The initial round of micro experiments targeted SPDK without deduplication, as the last category did. However, in this case, a more extensive set of configurations was used. In addition to the previous configurations, the hotspot access pattern, four processes, and the Kernels distribution were used for these experiments.

As to calculate the overhead introduced by deduplication, the second round of micro experiments used the memory version of the system. This version, since it is the most optimized in terms of performance, allowed for calculating the overhead introduced by inline and offline deduplication without the I/O overhead introduced by metadata persistence. Furthermore, in order to observe the interference that the background process introduces, two configurations were tested. At first, offline deduplication without the background process was examined. Next, the background process without delay between dirty blocks was enabled.

The last round of micro experiments targeted the persistent version of the system. This round allowed for calculating the overhead introduced by persistent metadata storage. In this one, besides evaluating the same mechanisms as before, the Feedback Loop Controller was also tested.

To sum up, the micro experiments targeted SPDK without deduplication, the memory implementation, and the system's persistent version. Inline and offline deduplication (with and without background processing) were tested individually, and in the persistent version, the Feedback Loop Controller was also examined.

5.3.1 *Results Analysis*

In this section, we introduce and analyze the results of the micro experiments. First, we start by presenting the results of read operations, and then we focus on write operations.

5.3.1.1 *Read Operations*

Mode	**nProcesses**	**Distribution**	**Throughput (MB/s)**		**Latency (us)**	
			Mean	**SDeviation**	**Mean**	**SDeviation**
Sequential	1	HighPerf	1477,58	6,42	3,00	0,00
		Kernels	1472,31	7,16	3,00	0,00
	4	HighPerf	1797,44	85,94	8,67	1,23
		Kernels	1793,65	17,47	8,75	1,06
Uniform	1	HighPerf	51,14	0,10	76,00	0,00
		Kernels	50,31	0,14	77,33	0,58
	4	HighPerf	182,08	0,34	85,50	3,00
		Kernels	178,56	0,46	87,25	1,76
HotSpot	1	HighPerf	196,07	0,09	20,00	0,00
		Kernels	187,27	1,07	21,00	0,00
	4	HighPerf	685,52	4,22	22,50	0,67
		Kernels	636,68	1,85	24,25	0,45

Table 5: Read Operations without Deduplication

Analyzing Table 5, which presents read operations without deduplication, a clear difference is observed among disk access patterns. While sequential operations reach 1475 MB/s, hotspot mode only achieves 190 MB/s, and uniform reads stay at 50 MB/s. Again, random reads are the bottleneck of the type of disk used at the experiments, with much higher overheads than sequential operations. On the other hand, hotspot mode only conducts its operations in a limited number of logical addresses. Such fact allows for the use of system caches, which are impossible to take advantage of in uniform mode. At last, sequential reads present the higher throughout, as expected.

Regarding the number of processes used, 1 and 4, it is quickly noticeable that higher values are achieved with the latter. For example, sequential operations raise from 1475 MB/s to 1795 MB/s, uniform from 50 MB/s to 180 MB/s, and hotspot performance increases from 190 MB/s to 630 MB/s.

Despite the increase seen across the board, it is clear that it is not proportional to all access patterns since sequential performance only increases around 20%, while uniform and hotspot report gains of around 3.5 times. Such a difference is caused by the fact that, since sequential reads perform very well, they use almost all disk capability. In the other two patterns, as they do not use a significant amount of the available bandwidth, higher concurrency leads to an almost proportional increase in throughput.

Finally, it is noticeable that the distribution used does not influence the results. Such a fact was expected as all blocks were stored on disk due to the lack of deduplication.

Mode	**nProcesses**	**Distribution**	**Throughput (MB/s)**		**Latency (us)**	
			Mean	**SDeviation**	**Mean**	**SDeviation**
Sequential	1	HighPerf	325,96	4,26	12,00	0,00
		Kernels	392,40	0,50	10,00	0,00
	4	HighPerf	301,45	0,78	52,00	0,00
		Kernels	490,89	0,19	32,00	0,00
Uniform	1	HighPerf	37,87	0,40	102,67	1,15
		Kernels	39,37	0,05	99,00	0,00
	4	HighPerf	138,75	0,33	112,33	0,78
		Kernels	144,36	0,39	108,00	1,48
HotSpot	1	HighPerf	114,04	3,07	34,00	1,00
		Kernels	130,33	1,06	30,00	0,00
	4	HighPerf	368,74	3,41	42,25	0,62
		Kernels	417,75	23,26	34,92	0,90

Table 6: Read Operations with Memory Deduplication

Examining Table 6, we can note that the results between operation modes follow the same pattern as before.

On the other hand, since deduplication is now active, the distribution used can interfere with the performance. Focusing on the results, we can observe that sequential performance increases from 325,96 MB/s when using highperf to 392,40 MB/s with kernels. Also, uniform reads performance improves from 37,87 MB/s to 39,37 MB/s, and hotspot operations reach 130,33 MB/s from 114,04 MB/s.

These differences are directly related to the number of duplicates introduced in the system by each. Actually, as the kernels distribution presents a substantial amount of redundant information, the volume of data actually stored on the disk is very scarce. The previous fact allows for previous reads to be kept in cache, lowering the overall disk usage.

When comparing the use of 1 and 4 processes, we can observe that uniform performance increased from 38 MB/s to 140 MB/s and that the hotspot mode also recorded substantial gains (120 MB/s to around 400 MB/s). However, sequential performance with the highperf distribution suffered a small drop to 301,45 MB/s compared to the 325,96 MB/s that had been achieved with a single process. Once again, the substantial amount of unique data makes caching impossible, requiring accessing the disk for every request. These accesses from four different processes caused interference with each other, which ultimately lead to performance drops.

In order to perceive the influence of deduplication in the overall system performance, we must turn our attention to both tables. After comparing its results, a performance decrease is evident. For example, sequential operations dropped from 1475 MB/s to 300/400 MB/s, uniform performance decreased to 38 MB/s from 50 MB/s, and hotspot reads declined from 190 MB/s to 120 MB/s.

Because a new layer of processing was added to the workflow, a reduction in throughput was expected. However, the drop is considerably higher in sequential reads when compared to the others. Despite the extra processing costs seen in uniform and hotspot, deduplication also introduces disk fragmentation, most visible in sequential requests. While sequential logical addresses translate to sequential physical addresses in a typical environment, such a statement is not accurate with deduplication since sequential logical addresses can be associated with physical blocks scattered across the disk.

Mode	**nProcesses**	**Distribution**	**Throughput (MB/s)**		**Latency (us)**	
			Mean	**SDeviation**	**Mean**	**SDeviation**
Sequential	1	HighPerf	249,65	0,60	15,67	0,58
		Kernels	336,06	0,50	11,33	0,58
	4	HighPerf	233,94	0,59	66,67	0,49
		Kernels	449,28	0,30	35,00	0,00
Uniform	1	HighPerf	35,71	0,08	109,00	0,00
		Kernels	36,98	0,03	105,00	0,00
	4	HighPerf	126,00	0,30	123,67	0,49
		Kernels	132,97	0,24	117,00	0,00
HotSpot	1	HighPerf	132,06	0,59	29,00	0,00
		Kernels	137,18	0,35	28,00	0,00
	4	HighPerf	424,31	2,54	36,42	0,51
		Kernels	463,60	14,39	33,33	0,98

Table 7: Read Operations with Persistent Deduplication

Considering that the memory implementation is always the most efficient version of a system, it was evident that the switch to persistent storage would bring performance losses. This decrease in performance can be confirmed in Table 7. These results show a sequential performance of around 250/330 MB/s, compared to the 325/390 MB/s with the memory version. Uniform reads also dropped from 38 MB/s to 36 MB/s.

Since it focuses its operation on a limited number of addresses, the hotspot mode can, once again, take full advantage of data already cached. As it relies on caching mechanisms, queries to leveldb are minimal in this operation mode, which allows the performance to remain stable between the two versions.

Environment	**CPU (%)**	**Memory (GB)**
No Deduplication	9,79	4,98
Memory Deduplication	43,69	9,21
Persistent Deduplication	42,54	3,57

Table 8: Resources

It is possible to note, from analyzing Table 8, which compares the resources used by the multiple testing environments, an increase in CPU (from 9,79% to 43,69%) and memory usage (from 4,98 GB to 9,21 GB) between the environment without deduplication and HIODS memory version. Actually, the increase was expected as deduplication introduces additional processing. Furthermore, as the metadata structures were stored in memory, such an increase was also foreseen.

Finally, the processing power used by the persistent version of HIODS remains unchanged, as the process itself does not change between versions. However, the metadata is now stored in leveldb, which allows for freeing the memory previously used to store the auxiliary data.

5.3.1.2 *Write Operations*

Mode	**nProcesses**	**Distribution**	**Throughput (MB/s)**		**Latency (us)**	
			Mean	**SDeviation**	**Mean**	**SDeviation**
Sequential	1	HighPerf	441,78	10,51	7,33	0,58
		Kernels	451,66	1,37	7,00	0,00
	4	HighPerf	435,19	4,00	34,33	0,49
		Kernels	439,68	1,91	34,00	0,00
Uniform	1	HighPerf	400,76	24,74	8,00	1,00
		Kernels	387,36	1,10	8,00	0,00
	4	HighPerf	389,18	3,53	38,67	0,49
		Kernels	392,61	5,82	38,33	0,49
HotSpot	1	HighPerf	380,17	11,37	8,67	0,58
		Kernels	378,71	6,91	8,67	0,58
	4	HighPerf	392,64	1,25	38,17	0,39
		Kernels	402,07	2,29	37,33	0,49

Table 9: Write Operations without Deduplication

Concerning write operations and observing Table 9, it is possible to conclude that the sequential mode delivered the best performance among the three, achieving around 440 MB/s. In fact, such results were expected as sequential operations introduce the lowest overheads. Changing the focus to uniform and hotspot operations, a slight decrease to around 390 MB/s is observed on both. Actually, modern hardware like regular SSDs and NVMe SSDs can reach almost identical performances between sequential and uniform write operations as they do not possess physical components like regular HDDs, where uniform operations are significantly more costly.

Moving on to the differences between 1 and 4 processes, stability in the results is observed. Contrary to what happened in the readings, a single process can take full advantage of the available bandwidth. With the introduction of multiple processes, the disk resources must be shared between them, which can cause interference among the processes, occasionally leading to slightly lower performances.

Finally, the content distribution used does not influence the results because deduplication is not implemented on this first setup.

Mode	nProcesses	Distribution	Throughput (MB/s)		Latency (us)	
			Mean	SDeviation	Mean	SDeviation
Sequential	1	HighPerf	293,89	1,88	12,00	0,00
		Kernels	296,41	1,22	12,00	0,00
	4	HighPerf	286,01	2,20	53,00	0,00
		Kernels	294,55	0,63	51,33	0,49
Uniform	1	HighPerf	271,13	0,70	13,00	0,00
		Kernels	276,79	0,74	12,00	0,00
	4	HighPerf	282,83	3,07	53,83	0,39
		Kernels	284,39	0,91	53,17	0,39
HotSpot	1	HighPerf	461,10	2,71	7,00	0,00
		Kernels	465,89	2,79	7,00	0,00
	4	HighPerf	443,10	4,23	33,42	0,51
		Kernels	446,22	0,91	33,00	0,00

Table 10: Write Operations with Inline Memory Deduplication

Following the base measurements, a memory deduplication layer was introduced. With this version, the overhead introduced by deduplication will be easily recognized. Firstly, the inline mode was tested, and the results are shown in Table 10.

Focusing on these last results, a general performance drop compared to the environment without deduplication is quickly perceived. For instance, sequential write performance decreased from 440 MB/s to 290 MB/s, and uniform mode performance dropped to 270/280 MB/s from the 390 MB/s previously obtained. The drop in performance is a direct consequence of an additional processing layer.

A second observation is a reduced difference between sequential and uniform operations, which was around 50 MB/s without deduplication, and now stands at 15 MB/s. Actually, when started from scratch, the deduplication system transforms all other requests into sequential ones as sequential addresses are provided by FreeBlocks to the Deduplication Engine. The previous fact, combined with operating system optimizations and lower disk usage, as it is not necessary to store all data, allow hotspot mode to gain 20% in performance, delivering around 450 MB/s versus the original 390 MB/s.

Directing our attention to the last table alone, slightly higher results are observed when using the kernels content distribution, when compared to the highperf one. For example, sequential operations using one process and the highperf content distribution delivered 293,89 MB/s, while the same configuration using the kernels content distribution reached 296,41 MB/s. In fact, the higher the number of duplicates, the better the engine will perform since fewer interactions with the components and fewer communications with the disk are performed. With 75% of redundant content against only 20%, kernels does not produce such heavy disk usage, which, combined with fewer interactions with the components, results in higher performance.

Finally, we can still observe that the competition for resources among the four processes leads to small decreases compared to a single process.

Mode	nProcesses	Distribution	Throughput (MB/s)		Latency (us)	
			Mean	SDeviation	Mean	SDeviation
Sequential	1	HighPerf	307,33	0,17	11,00	0,00
		Kernels	307,82	0,12	11,00	0,00
	4	HighPerf	306,63	0,25	49,17	0,39
		Kernels	307,04	0,61	49,08	0,29
Uniform	1	HighPerf	295,88	0,05	12,00	0,00
		Kernels	297,07	0,33	11,00	0,00
	4	HighPerf	293,98	0,44	51,33	0,49
		Kernels	295,58	0,04	51,00	0,00
HotSpot	1	HighPerf	481,43	0,51	6,00	0,00
		Kernels	490,32	1,43	6,00	0,00
	4	HighPerf	467,40	11,64	31,75	0,75
		Kernels	459,60	20,52	32,17	1,40

Table 11: Write Operations with Offline Memory Deduplication without Background Processing

Inline deduplication introduces extra overhead since it finds and removes duplicate data in the I/O critical path. As an alternative, offline deduplication was developed. This new mode does not find or remove duplicate data in the I/O critical path, relying on a background job that competes for resources with the main task.

To observe the real impact that the background process has on performance, tests targeting offline deduplication without and with it were conducted. Table 11 presents the results of the former.

Since the targeted environment does not find or remove duplicates, either in the critical path or later in time, a boost in performance was expected and is confirmed by the results. For instance, sequential operations that achieved 290 MB/s with inline deduplication, reached 307 MB/s. Uniform and hotspot performance also improved from 275 MB/s and 450 MB/s to 295 MB/s and 475 MB/s, respectively.

Mode	nProcesses	Distribution	Throughput (MB/s)		Latency (us)	
			Mean	SDeviation	Mean	SDeviation
Sequential	1	HighPerf	294,69	0,28	12,00	0,00
		Kernels	294,76	0,42	12,00	0,00
	4	HighPerf	293,46	0,62	51,83	0,39
		Kernels	294,41	1,33	51,58	0,51
Uniform	1	HighPerf	276,10	0,28	12,00	0,00
		Kernels	277,08	0,44	12,00	0,00
	4	HighPerf	273,11	0,44	55,25	0,45
		Kernels	274,99	0,58	55,00	0,00
HotSpot	1	HighPerf	455,65	2,62	7,00	0,00
		Kernels	458,28	3,34	7,00	0,00
	4	HighPerf	434,81	7,31	34,25	0,62
		Kernels	467,51	3,60	31,67	0,49

Table 12: Write Operations with Offline Memory Deduplication with Background Processing

Despite exhibiting the best possible performance, the targeted HIODS setup in Table 11 does not find or remove duplicates because offline deduplication requires a background job to do so. Therefore, the background task was activated, and the results are presented in Table 12.

As expected, since the secondary job competes for resources with the primary process, a performance drop is noticeable. In fact, the sequential mode that delivered 307 MB/s without the background job could only achieve 294 MB/s with the current system configuration. The same drop was experienced in the remaining modes, which saw their performance drop to 275 MB/s and 450 MB/s from the 295 MB/s and 475 MB/s previously obtained by uniform and hotspot experiments.

Mode	nProcesses	Distribution	Throughput (MB/s)		Latency (us)	
			Mean	SDeviation	Mean	SDeviation
Sequential	1	HighPerf	77,02	0,82	49,33	0,58
		Kernels	95,39	0,33	39,00	0,00
	4	HighPerf	77,69	0,15	198,33	0,58
		Kernels	96,05	0,11	160,67	0,58
Uniform	1	HighPerf	61,25	0,22	62,00	0,00
		Kernels	81,71	0,17	46,00	0,00
	4	HighPerf	60,83	0,04	253,33	1,15
		Kernels	81,11	0,21	190,00	0,00
HotSpot	1	HighPerf	102,05	22,39	32,00	0,00
		Kernels	153,71	2,85	23,33	0,58
	4	HighPerf	133,15	0,31	113,33	1,53
		Kernels	153,62	0,53	99,33	0,58

Table 13: Write Operations with Inline Persistent Deduplication

The previous round of tests targeted the memory version of the system. However, a system should not rely on memory to store its data. Therefore, Table 13 reveals the results from the tests performed with inline persistent deduplication.

When comparing the memory version (Table 10) with the persistent version (Table 13), it is possible to realize the negative impact that a key-value store like leveldb has on performance. In the former, sequential and uniform performance hovered around 250 to 300 MB/s, and hotspot operations reached 450 MB/s. With these last results, the first two modes dropped to between 60 and 100 MB/s, and hotspot experiments delivered higher results between 100 and 150 MB/s, but still far from the original performance.

Furthermore, the performance disparity between highperf and kernels content distributions, which was around 5 MB/s in memory experiments, is now close to 20 MB/s. Besides the reduced disk accesses when using the latter, the former has substantially more unique data that must be stored on the Index and Reverse Index, which leads to more leveldb interactions, thus significantly increasing the overhead with the highperf distribution.

Mode	nProcesses	Distribution	Throughput (MB/s)		Latency (us)	
			Mean	SDeviation	Mean	SDeviation
Sequential	1	HighPerf	143,13	0,30	26,00	0,00
		Kernels	143,53	0,30	26,00	0,00
	4	HighPerf	143,62	0,18	106,67	0,58
		Kernels	143,02	0,28	107,00	0,00
Uniform	1	HighPerf	139,81	0,10	26,00	0,00
		Kernels	139,99	0,38	26,00	0,00
	4	HighPerf	139,88	0,21	109,33	0,58
		Kernels	140,44	0,14	109,00	0,00
HotSpot	1	HighPerf	261,00	0,29	13,00	0,00
		Kernels	263,13	1,22	13,00	0,00
	4	HighPerf	263,40	1,10	57,00	0,00
		Kernels	262,46	5,32	57,33	1,53

Table 14: Write Operations with Offline Persistent Deduplication without Background Processing

As to observe the real overhead introduced by inline mode, since the system does not live in memory anymore, tests targeting persistent offline deduplication without background processing were executed once again. The results can be seen in Table 14.

As expected, a performance boost that reached two times the inline performance is witnessed. For example, sequential operations that achieved around 80 MB/s now reach 143 MB/s. Furthermore, uniform performance also increased from about 71 MB/s to 140 MB/s, and the 100 to 150 MB/s hotspot operations now reach 262 MB/s.

Such gains are directly related to the fact that offline mode does not find or remove duplicates in the I/O critical path. However, while the gains were almost negligible when in memory, here, the performance increase reached 50%. These considerable gains relate to the difference in operations between inline and offline, which were not best seen in memory due to its high-speed nature. However, when on leveldb, every operation involving it causes additional overhead, resulting in a much lower performance with inline deduplication.

Regarding the results involving the two different distributions, it is clear that the previously seen disparity no longer exists. The gap disappearance relates to the fact that all blocks are written to disk with offline deduplication. Furthermore, as the current target does not find or remove duplicates, the Index and Reverse Index are only consulted for update operations resulting in almost negligible overheads. The existence of updates in uniform mode is also the reason for its slightly lower results when compared to sequential operations.

Mode	nProcesses	Distribution	Throughput (MB/s)		Latency (us)	
			Mean	SDeviation	Mean	SDeviation
Sequential	1	HighPerf	60,98	0,61	62,33	0,58
		Kernels	81,07	0,38	46,33	0,58
	4	HighPerf	67,94	2,29	227,67	7,77
		Kernels	83,46	0,75	185,33	1,53
Uniform	1	HighPerf	47,56	0,23	80,33	0,58
		Kernels	71,38	1,02	53,00	1,00
	4	HighPerf	44,41	0,37	346,67	2,08
		Kernels	71,24	0,52	215,33	2,52
HotSpot	1	HighPerf	122,25	1,42	30,33	0,58
		Kernels	136,87	0,74	27,00	0,00
	4	HighPerf	114,27	1,02	134,33	1,15
		Kernels	129,43	2,26	117,00	2,00

Table 15: Write Operations with Offline Persistent Deduplication with Background Processing

One of the most significant downsides of offline deduplication is the need for extra resources. In addition to storage space, since all chunks are stored in the disk, offline deduplication also requires a background job to find and remove duplicates, which competes for resources alongside the foreground job. Table 15 presents the results of offline persistent deduplication with simultaneous background processing.

As already seen in the memory version, although more lightly, a 50% drop in performance from the previous results is discerned. For example, sequential performance dropped from 143 MB/s to around 70 MB/s, uniform operations decreased from 140 MB/s to an average of 58 MB/s, and hotspot experiments dropped to 125 MB/s from 263 MB/s.

In reality, the competition for resources is so intense that even inline deduplication with all its overhead outperforms it. For example, sequential mode, which reported 75 to 100 MB/s with inline deduplication, only achieves 60 to 80 MB/s with the current setup.

A second difference with the previous results is the return of the contrast between distributions that did not exist without background processing. In reality, highperf introduces more unique data into the system, which must be stored on Index and Reverse Index, significantly increasing leveldb accesses in the background job. As such operations are quite costly, the processing power is redirected to the secondary task decreasing the overall system capability.

5.3.1.3 *Feedback Loop Controller*

Unlike inline deduplication, where redundant data is found and eliminated directly in the I/O critical path, offline mode requires a secondary process to achieve the same goal.

However, as the secondary job also requires resources to find and eliminate duplicate data, the total system capacity must be shared between the two. This split of resources may cause a decrease in the primary process performance, which we witnessed in the previous experiments (Table 15).

As to reduce the background process activity, which releases resources to the primary job, increasing its overall performance, HIODS implements a Feedback Loop Controller mechanism capable of regulating the secondary job throughput, thus helping to reach and maintain the performance goal.

Mode	nProcesses	Distribution	Throughput (MB/s)		Latency (us)	
			Mean	SDeviation	Mean	SDeviation
Sequential	1	HighPerf	117,98	0,62	31,67	0,58
		Kernels	119,89	1,65	30,67	0,58
	4	HighPerf	118,29	0,41	129,67	0,58
		Kernels	120,34	0,04	127,00	0,00
Uniform	1	HighPerf	123,03	0,25	30,00	0,00
		Kernels	123,41	0,26	30,00	0,00
	4	HighPerf	123,66	0,47	124,00	1,00
		Kernels	124,17	0,26	123,67	0,58
HotSpot	1	HighPerf	241,22	2,21	14,33	0,58
		Kernels	248,61	1,53	14,00	0,00
	4	HighPerf	247,50	0,61	60,33	0,58
		Kernels	250,84	0,13	60,00	0,00

Table 16: Write Operations with Offline Persistent Deduplication with Feedback Loop Controller

Analyzing the results presented in Table 16, we can conclude that this FLC mechanism, capable of limiting the background job throughput, is vital in helping HIODS reach and maintain the desired performance. In fact, the sequential and uniform results without the FLC achieved between 60 to 80 MB/s but, with the current strategy, measurements between 118 and 125 MB/s were reached.

Despite the good results, the tests show that the FLC is not being aggressive enough because it did not reach the goal set to 35000 operations per second (136,72 MB/s) with the following parameters: $k_p = 5.0$, $k_i = 0.001$, and $k_d = 0.25$. However, fixing the lack of aggressiveness is a relatively easy task since it only requires some adjustments to its parameters.

Regarding the processing power and memory used by these experiments, similar values to the read tests (Table 8) were measured. Furthermore, both inline and offline modes were able to save around 11.5 GB of storage space when highperf was used, and 45.8 GB with the kernels content distribution.

5.4 MACRO EXPERIMENTS

This last series of tests had as primary objective to analyze how HIODS reacts to changes in the workload with distinct performance goals. In order to examine such reactions, the mechanism for switching between deduplication modes, and the FLC were observed.

In order to achieve the proposed goals, four experiments were conducted: Mountain, Valley, Ascending Stairs, and Descending Stars. All the previous defined multiple performance goals that were increased or decreased after writing 64 GB at a fixed pace to simulate workload changes. For these experiments, DEDISbench nominal flag, which allows for defining workload at fixed rates, was used.

5.4.1 *Mountain/Valley*

With the objective to simulate two distinct workloads, Mountain/Valley experiments defined two performance tiers: a relatively low, which allowed the overhead introduced by inline mode, and a much higher second tier that made offline deduplication the only option. The chosen tiers were 4000 and 20000 operations per second.

Therefore, mountain mode started with a 64GB write at 4000 operations per second. Following that first operation, an operation of the same size was realized at 20000 operations per second. Finally, a third 4000 operation per second 64GB write was conducted. On the other hand, the Valley mode performed a first 64GB write at 20000 operations per second, which then was decreased to 4000 operations a second 64GB write. Finally, a third write similar to the first was performed.

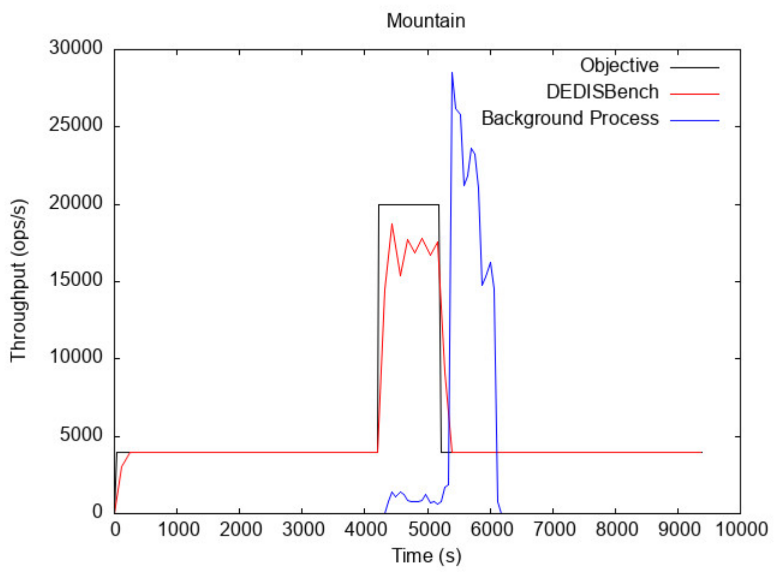

Figure 9: Mountain Test

The Mountain results are shown in Figure 9 under a plot. In it, it is possible to see the throughput achieved by DEDISbench (red line), which corresponds to HIODS performance, and it is expected to match the target performance (black line). Furthermore, to watch the FLC mechanism in action, the background job throughput is also shown (blue line). These colors represent the same measures in all the following figures. Focusing on the results, three phases are clearly seen.

At the beginning of the test, phase 1, the engine started with inline deduplication active and, as expected, was able to maintain that operation mode until the end of this phase. The null background processing throughput is evidence of such achievement.

The second set of operations, which represents the mountain, had the primary goal of activating offline deduplication through a high enough desired performance goal. As observable by the graphic, the background job started processing the undealt data at around 4300 seconds, indicating that the switch to offline mode was triggered. Finally, by the end of this phase, difficulty in achieving the desired goal lead the FLC to increase the delay, lowering the background job throughput to around 600 operations per second, from close to 1000 operations per second.

At last, the switch back to inline mode was intended with the last set of write operations. When switching from offline to inline mode, the Controller disables the FLC mechanism removing the delay in the processing of undealt blocks, which leads to a mass processing of the DirtyQueue, reaching 25000 operations per second. When analyzing the last phase, an exponential increase in the background job throughput clearly shows that the intended switch was made.

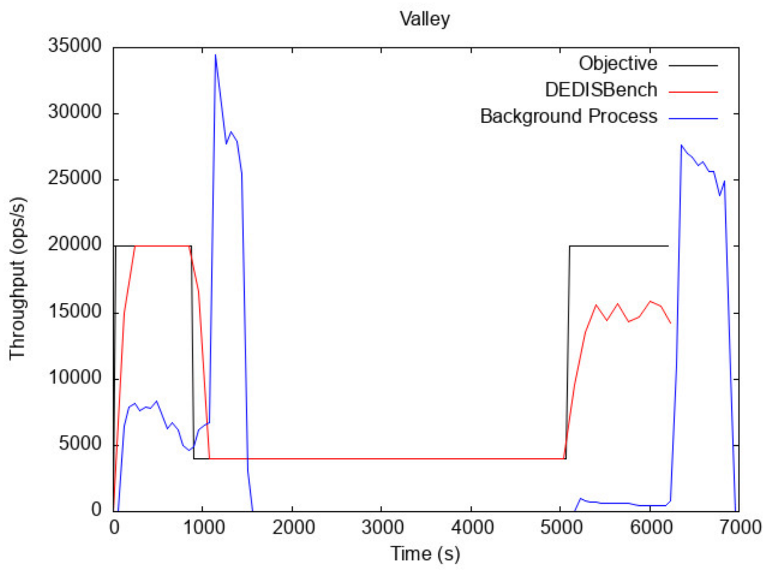

Figure 10: Valley Test

While the plot in Figure 9 shows the mountain, Figure 10 presents the opposite test, the valley. So, this second test started with a high performance goal, which posteriorly slowed down. In the end, the high objective was increased to motivate the switch to offline mode.

This second test started with offline mode and a goal of 20000 operations per second, which was achieved as witnessed in the plot above. However, at the end of this first set of operations, difficulties in maintaining the desired objective lead the FLC to increase its delay, which lowered the background processing throughput from 7500 to 5000 operations per second, allowing the foreground process to keep performing at the desired level.

The second round of operations was intended at 4000 operations per second, which should trigger the switch to inline mode. As already explained in the previous results, the exponential growth of the background job throughput shows that the switch occurred.

Finally, the desired goal was once again increased in the final phase, making the switch to offline deduplication necessary. As seen in the last phase, the background job starts to process data at around 5100 seconds, indicating such trade. Another interesting fact is the difference between performances achieved by the background job in the first and last phase. A significant decrease, from 7500 to 450 operations per second, is noticeable from the former to the latter, which indicates difficulties in achieving the desired performance objective, which was not fully reached. Such performance difference between the two phases is due to the use of resources to accommodate leveldb caches and other mechanisms, which gradually decreases the system performance and stability.

Finally, when the benchmark stopped issuing requests, at second 6300, the Deduplication Controller detected that the engine was not under stress anymore, switching back to inline mode, which led to the massive processing of dirty blocks.

5.4.2 *Stairs*

The second category of macro experiments had a slightly different configuration. Unlike the previous tests, where the switching mechanism was in the spotlight, with these experiments, the goal was to watch the FLC adjust the DirtyQueue delay accordingly to the current workload and system capacity.

With that objective in mind, this test set required an additional tier compared to the Mountain/Valley tests. Therefore, three performance tiers were established. Like the previous tests, the first was set to 4000 operations per second to activate inline deduplication. The next tier, set to 16000 operations per second, was meant for activating offline deduplication without a significant delay from the FLC. Finally, to lower the background job throughput to its minimum, the last tier was set to 20000 operations per second.

The results from ascending stairs (4-16-20) and descending stairs (20-16-4) are shown in Figures 11 and 12.

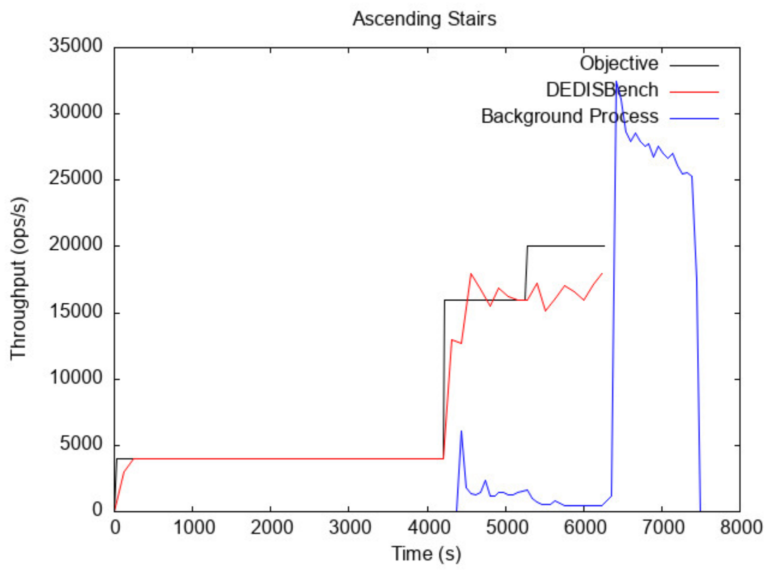

Figure 11: Ascending Stairs Test

Analyzing the plot from the ascending stairs test, similarities with Figure 9 are evident. In fact, also here the system started with inline mode and maintained it through the first 64GB.

In the second phase of this test, the desired performance goal was set to 16000 operations per second at around 4100 seconds. As expected and evidenced by the start of the simultaneous background processing, the offline mode was activated. In this phase, the background throughput averaged 1500 operations per second.

Next, the goal was set to 20000 operations per second. Due to the previously explained reasons, the goal was not often reached despite the FLC effort, which substantially increased the processing delay of undealt blocks in order to decrease the interference with the primary process. The much lower background job throughput, around 450 operations per second by the end, evidences the effort mentioned above compared to the previous phase.

At last, with the end of write operations at second 6400, the Deduplication Controller once again detected the lack of stress switching the operation mode to inline, leading to the massive processing of the DirtyQueue.

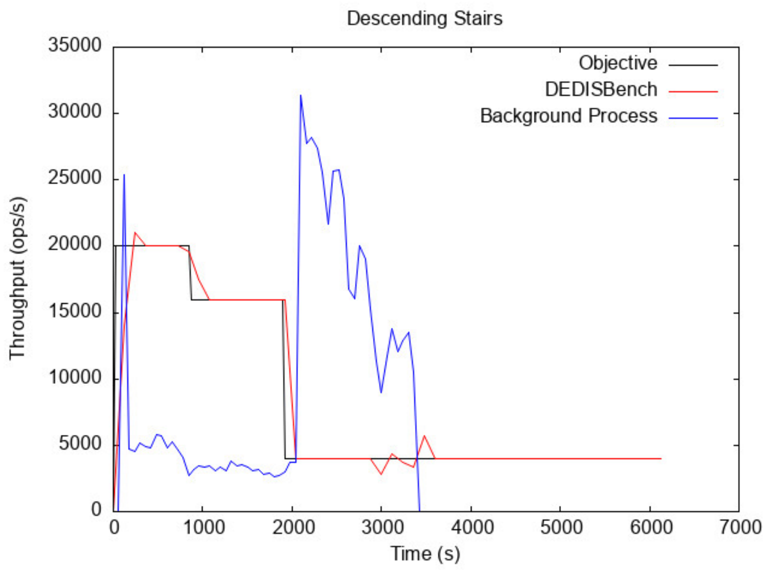

Figure 12: Descending Stairs Test

The final performed test reversed the previous operations. First, the engine was started with offline mode, and a 64GB write at 20000 operations per second was conducted. As evident in Figure 12, the desired goal was achieved with a background throughput at around 5000 operations per second. However, in the final stage of this first phase, the FLC saw the need to increase the delay. By reducing the background throughput to around 3500 operations per second, the foreground job was able to maintain the performance goal.

After the first 64GB, the goal was decreased to 16000 operations per second, which were delivered. This reduction should allow for a slight increase in background performance to occur. We can witness that improvement, from 3500 to 3750 operations per second, between the seconds 1000 and 2000.

Finally, the desired goal was set to 4000 operations per second which were delivered. As expected, around the second 2000, we witness massive processing of dirty blocks, indicating that the switch to inline deduplication occurred.

5.5 DISCUSSION

The performed tests had several distinctive objectives. With the first category, preliminary experiments, it was possible to compare the raw NVMe performance against a baseline implementation using SPDK. Micro experiments had a primary goal to observe the performance impact of deduplication. Furthermore, this second category also allowed for assessing individual system components. Finally, macro experiments targeted the deduplication mode switching mechanism and the Feedback Loop Controller.

From a general perspective, the first category show slightly lower results with SPDK, where the block-device is exported through nbd, than the tests conducted directly on the NVMe. These results were predictable since introducing additional processing layers leads to extra latency.

With the next category, micro experiments, it is possible to conclude several things. First, it is possible to observe the weight of deduplication in the overall system performance by comparing the system without deduplication with its memory version. In the same category, HIODS persistent version was tested. The results show a significant performance drop since we are now storing the Index, Reverse Index, and Metadata on disk using leveldb, which has much higher latencies than RAM. These results also show a more visible difference between inline and offline (with and without the background job) deduplication. In fact, offline deduplication without the background job presents significantly higher performance than inline deduplication. However, since it does not find or removes duplicates, a background job is necessary, which competes for resources with the foreground I/O task. The introduction of such a process leads to lower results than inline deduplication.

In order to control the background processing throughput, HIODS introduces a Feedback Loop Controller capable of delaying the processing of dirty blocks. The results of the tests conducted on this mechanism show that it was able to significantly increase performance, and improve it over inline deduplication, by reducing the resources needed by the background job.

Finally, the last category, macro experiments, focused on the switching mechanism and the Feedback Loop Controller. These results show that HIODS successfully changed its operation mode based on the system workload and performance objective. Furthermore, it was also possible to conclude that the FLC mechanism plays a vital role in helping HIODS reach and maintain the desired performance.

6 CONCLUSION

This dissertation presents HIODS, a hybrid inline and offline deduplication system capable of dynamically switching between inline and offline modes to achieve and maintain the desired performance goals of distinct applications.

When analyzing the state of the art it is possible to conclude that some systems already leverage both deduplication modes. However, these systems present new algorithms and mechanisms whose main objective is only to mitigate the overhead introduced by inline deduplication. For example, DIODE and D3 classify the input file into three types based on its extension degree of deduplication. Such classification is then used to determine the best deduplication mode for the file. Therefore, in a general fashion, these systems identify the best candidates to inline deduplication, leaving the remaining ones, which may cause extra performance interference, to be dealt in background.

The deduplication system designed in this dissertation aims differently. While the previous systems decide the best operation mode according to each request's data to efficiently perform inline deduplication, HIODS bases its decision on the performance goals of applications using the storage system to choose the best deduplication mode. In order to achieve the proposed goal, the system relies on a performance objective that must be achieved and maintained to keep inline deduplication operating. In the cases where the overhead introduced by the inline mode does not allow achieving the proposed objective, the offline mode is chosen. Furthermore, HIODS also introduces a second mechanism based on Feedback Loop Controllers that limits the background deduplication job to reduce the interference with critical I/O operations and, again, achieve the desired application performance.

Following the design of the system, a prototype was implemented using SPDK. This prototype implements both inline and offline deduplication, the switching mechanism, and the Feedback Loop Controller (FLC) capable of rate limiting offline deduplication. To summarize these components, inline deduplication processes the storage request in the I/O critical path introducing additional overhead but only storing unique copies. On the other hand, the offline mode only performs the required procedures for later processing, thus reducing to the minimum the overhead in the I/O critical path but storing all data, unique or duplicate. Periodically, the switching mechanism analyses the current and desired performances and decides if the operation mode should change or not. Finally, suppose the offline mode is active, and the performance goal is still not being reached. In that case, the FLC delays the background deduplication processing, reducing the competition for resources between deduplication and I/O operations and, therefore, increasing storage I/O performance.

In order to evaluate the mechanisms introduced in HIODS, a series of experiments were designed and executed. The results show that HIODS successfully changed its operation mode based on the system workload and performance objective. Furthermore, it was also possible to conclude that the FLC mechanism plays a vital role in helping HIODS reach and maintain the desired performance.

In conclusion, this dissertation introduces a system capable of dynamically changing its deduplication mode to reach and maintain the performance requirements of applications.

6.1 FUTURE WORK

Since deduplication introduces additional performance costs, finding and removing duplicates directly in the I/O critical path or preparing the data for later processing, the optimization of both operation modes allows for lower overheads and higher performance. Implementing mechanisms capable of taking advantage of data locality, or more complex indexes with lower access times are a few examples of possible optimizations.

A second improvement in the system would be developing an alternative to persistently store the deduplication metadata. As observable in the micro experiments, the transition from memory to a persistent implementation with leveldb brought significant performance overhead. A possible solution would be to store this metadata directly on a disk partition while using SPDK to do it.

Also, despite the promising results shown by the Feedback Loop Controller mechanism, the system may benefit from research regarding its parameters, k_p, k_i, and k_d. Finally, the switch from offline to inline deduplication causes a burst of background deduplication operations. This burst can interfere with the foreground I/O operations, so a new mechanism capable of limiting such processing may also be an asset to HIODS.

BIBLIOGRAPHY

[1] Abhishek Mukherjee, Alvin Afuang, Bill Rojas, Hugh Ujhazy, and Theresa Rago. Iot growth demands rethink of long-term storage strategies, says idc. Technical report, International Data Corporation, July 2020.

[2] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. Technical report, International Data Corporation, November 2018.

[3] 451 Research. 69% of enterprises will have multi-cloud/hybrid it environments by 2019, but greater choice brings excessive complexity. Technical report, 451 Research, New York (NY) and Las Vegas (LV), November 2017.

[4] Aayushi Vernika Das, Delisha Clair Sequeira, Gulshan Damini Patel, and V R Srividhya. A survey on deduplication techniques in cloud storage with cryptographic techniques. In *2017 International Conference on Pervasive Computing and Networking*. International Journal of Engineering Research & Technology (IJERT), 2017.

[5] Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. A survey of secure data deduplication schemes for cloud storage systems. *ACM Computing Surveys*, 49(4):1–38, February 2017.

[6] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, November 2012.

[7] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Comput. Sur. 47, 1, Article 11*, page 30 pages, may 2014.

[8] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, December 2017.

[9] William J Bolosky, Scott Corbin, David Goebel, and John R Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. Seattle, WA, 2000.

[10] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2004.

[11] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *FAST*, volume 2, pages 89–101, 2002.

[12] Bo Hong, Demyn Plantenberg, Darrell DE Long, and Miriam Sivan-Zimet. Duplicate data elimination in a san file system. In *MSST*, pages 301–314, 2004.

[13] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[14] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[15] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[16] Sean C Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *USENIX Annual Technical Conference*, pages 143–156, 2008.

[17] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 1–14, 2008.

[18] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *FAST*, volume 10, pages 225–239, 2010.

[19] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *Fast*, volume 12, pages 1–14, 2012.

[20] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Fast*, volume 9, pages 111–123, 2009.

[21] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–9. IEEE, 2009.

[22] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *USENIX annual technical conference*, 2011.

[23] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *FAST*, volume 11, pages 77–90, 2011.

[24] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *FAST*, pages 91–103, 2011.

[25] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in ssds: Model and quantitative analysis. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[26] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: A scalable secondary storage. In *FAST*, volume 9, pages 197–210, 2009.

[27] Tian-Ming Yang, Dan Feng, Zhong-ying Niu, and Ya-ping Wan. Scalable high performance de-duplication backup via hash join. *Journal of Zhejiang University SCIENCE C*, 11(5):315–327, 2010.

[28] Austin T Clements, Irfan Ahmad, Murali Vilayannur, Jinyuan Li, et al. Decentralized deduplication in san cluster file systems. In *USENIX annual technical conference*, pages 101–114, 2009.

[29] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–14, 2009.

[30] Randal C Burns and Darrell DE Long. Efficient distributed backup with delta compression. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 27–36, 1997.

[31] Lawrence You and Christos T Karamanolis. Evaluation of efficient archival storage techniques. In *MSST*, pages 227–232, 2004.

[32] Lawrence L You, Kristal T Pollack, and Darrell DE Long. Deep store: An archival storage system architecture. In *21st International Conference on Data Engineering (ICDE'05)*, pages 804–815. IEEE, 2005.

[33] John R Douceur, Atul Adya, William J Bolosky, P Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings 22nd international conference on distributed computing systems*, pages 617–624. IEEE, 2002.

[34] Yan Tang, Jianwei Yin, Shuiguang Deng, and Ying Li. DIODE: Dynamic Inline-Offline DEduplication providing efficient space-saving and read/write performance for primary storage systems. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, September 2016.

[35] 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*. IEEE, May 2014.

[36] Jianwei Yin, Yan Tang, Shuiguang Deng, Ying Li, and Albert Y. Zomaya. D3 : A dynamic dual-phase deduplication framework for distributed primary storage. *IEEE Transactions on Computers*, 67(2):193–207, February 2018.

[37] Huijun Wu, Chen Wang, Yinjin Fu, Sherif Sakr, Liming Zhu, and Kai Lu. HPDedup: A hybrid prioritized data deduplication mechanism for primary storage in the cloud, 2017.

[38] Amdewar Godavari, Chapram Sudhakar, and T. Ramesh. Hybrid Deduplication System—a block-level similarity-based approach. *IEEE Systems Journal*, pages 1–11, 2020.

[39] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six degrees of scientific data: reading patterns for extreme scale science io. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 49–60, 2011.

[40] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication—large scale study and system design. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.

[41] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, 2016.

[42] Network block device. https://nbd.sourceforge.io/. Accessed: 18-12-2020.

[43] Glib reference manual. https://developer.gnome.org/glib/. Accessed: 18-12-2020.

[44] Google. google/leveldb. https://github.com/google/leveldb. Accessed: 18-12-2020.

[45] Vineet Kumar, BC Nakra, and AP Mittal. A review on classical and fuzzy pid controllers. *International Journal of Intelligent Control and Systems*, 16(3):170–181, 2011.

[46] JTPaulo. jtpaulo/dedisbench. https://github.com/jtpaulo/dedisbench. Accessed: 18-12-2020.

[47] dstat. https://linux.die.net/man/1/dstat. Accessed: 18-12-2020.