Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bericht

Deduplication in JULEA

vorgelegt von

Julius Plehn

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	M. Sc. Informatik
Matrikelnummer:	6535163
Betreuer:	Dr. Michael Kuhn

Hamburg, 06. August 2021

Abstract

In general, deduplication can be used to only store a single copy of an entity that occurs several times by referencing all further entities to the initial one. In terms of file systems this means that the data is separated by some strategy and a unique identifier of this specific part is generated. Only if this identifier is new to the system a traditional write operation is performed and the identifier is stored. Whenever another part has the same content and therefore the same identifier a reference to the existing one is made. This strategy therefore can decrease the amount of physical storage needed while also introducing a few challenges regarding the separation, identification and storage of the identifiers required.

In this report several approaches to deduplication are presented and use cases and real world implementations are shown. Finally, deduplication is implemented into the JULEA storage framework. In the evaluation it is shown, that depending on the use case and the type of data significant storage savings are possible.

Contents

1 Introduction

In this chapter the general purpose of data reduction techniques and the relevance of deduplication in this regard is shown. Finally, an introduction into the basics of deduplication is given.

1.1 Motivation

On a file system level there are two concepts that are used regularly when data reduction is of interest. The first possibility is to use compression and this method is implemented in file systems like ZFS and Btrfs.

Using compression it is possible to reduce the size of the individual blocks by applying algorithms like LZ4, ZSTD and GZIP. This method works exceptionally well on many types of data except on some formats that apply reduction techniques by themselves like JPEG or already compressed files. Another benefit is, that only a few additional details about the compressed blocks has to be handled. This includes the used compression algorithm, their attributes and the physical size of the underlying data.

An alternative is to use deduplication. In the same way compression excels at some specific use cases deduplication performs especially well when the same data occurs multiple times throughout the dataset. This explicitly also includes already compressed data like images and videos. A disadvantage in this case is that much more additional metadata regarding the references to other data parts has to be stored.

Those two techniques do not exclude each other. The strengths of each concept can also be combined into an example I/O pipeline seen in Figure 1.1. In the first part deduplication is performed and the first and the last part of the file are identified to be the same. From a physical storage perspective the fourth block therefore does not has to be saved. Finally, the remaining chunks are compressed and therefore are reduced in size.

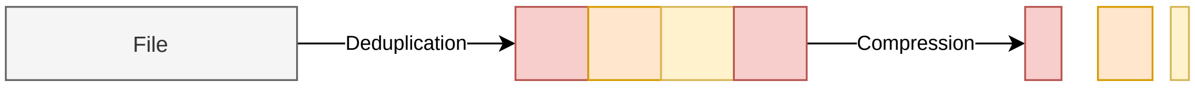


Figure 1.1: Example reduction pipeline

1.2 Deduplication

In this section an introduction into the basic principles of deduplication is given and the read and write processes are highlighted. This also lays the foundation for the implementation given in Chapter 4.

When using deduplication the overall goal is to reduce physical storage requirements by storing repetitive data only once and reusing those data pieces throughout the whole domain space. The main focus in this report is the integration into file systems and this process is shown in the remainder.

In Figure 1.2 an overview of a write and read operation on the same file is given. When writing a file the first step is to create chunks according to a specific policy. In this case static chunking is used, while maintaining a chunk size of 128 kB. Each chunk is then hashed using a cryptographic function with a sufficient certainty that only the same data chunk results in the same fingerprint. Using these fingerprints it is now possible to check if another chunk has previously been written which resulted in the same fingerprint. For every chunk that ocurred the associated reference counter of the fingerprint is incremented.

In this example the fingerprints are stored in a hash table and the first, second and sixth chunk referenced by the fingerprint `0x2` has once been used previously. This system now has the capability to decide that these chunks do not need to be saved again and therefore only those two new chunks have to be written. In this example only half of the physical storage is required in total. The last step is to update the reference counters in the metadata backend. Besides indicating duplicates of individual chunks the reference counter is also used for garbage collection purposes. When deleting a file all associated reference counters are decremented and when reaching zero the chunk is removed.

On the contrary when reading a file the first step is to use the metadata storage to retrieve a list of the chunks associated with the file name. Those identifiers can then be used to read the chunks independently and to recreate the file according to the physical layout. In order to reduce the amount of I/O required to read the file only unique chunks have to be received. The final file can be reassembled by utilizing the chunk layout provided by the metadata backend.

This top level view on deduplication serves as an introduction to this topic. As can be seen there are several ways to create chunking policies and various ways to implement fingerprinting, metadata handling and I/O paths. Further possibilities are shown in Chapter 2 and an implementation is shown in Chapter 4.

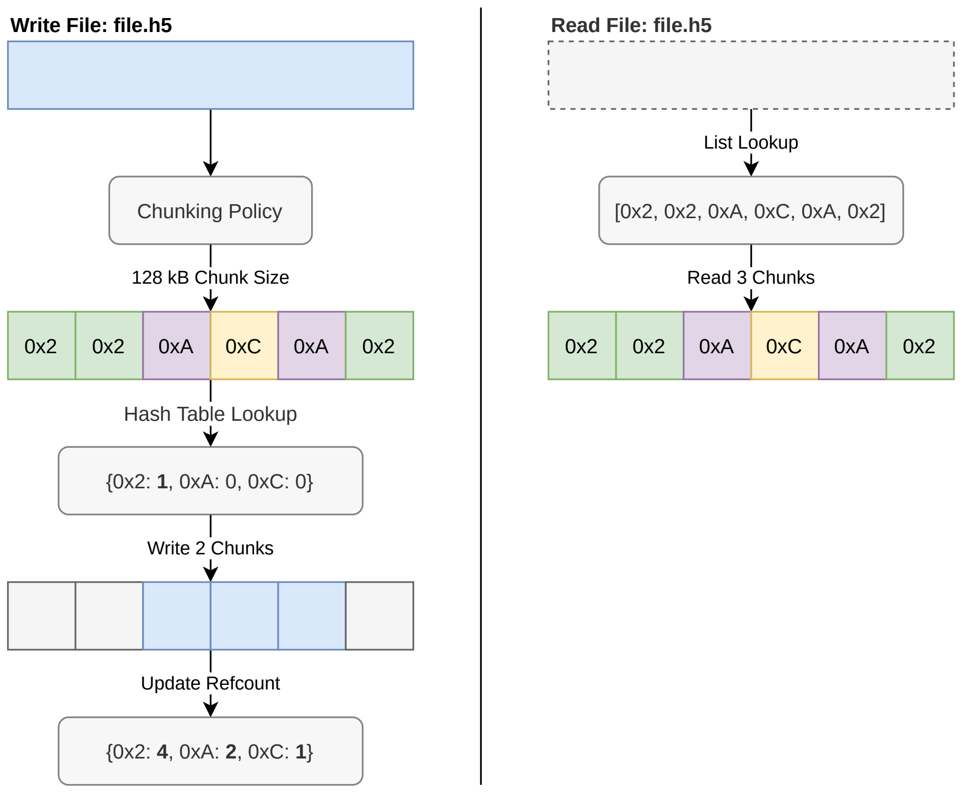


Figure 1.2: Deduplication Paths

2 Background

When talking about deduplication it is important to differentiate the possible use cases and how they influence the implementation. As deduplication can be used in local and distributed file systems, while also having unique I/O requirements, different techniques are used in practice and for research purposes. Those are explored in this section.

2.1 Chunking

The most notable differences are in the use of the chunking technique. Chunks can either have a static or a dynamic size. A prominent example of the use of static sizes can be found in ZFS where the chunk size of those fingerprinted blocks is aligned with the *recordsize*. The benefit of this approach is the relative ease of implementation and [1, p. 165] has shown reduced processing time in relation to dynamic chunk sizes. The deduplication ratio however, which is defined as $DR = \frac{\text{file size}}{\text{deduplication size}}$, is lower than with dynamic sizes and is subject to several disadvantages.

The biggest issue with static chunking and why dynamic chunking is preferred is the boundary shift problem. This issue arises when a small modification in a file leads to a rewrite of the remaining file as those static chunks change in the remainder of the file as well. In the worst case scenario this change occurs in the beginning of the file, which therefore would require a complete rewrite of the file. To conquer this issue dynamic or content-defined chunking uses chunks of variable length. This is achieved by the use of rolling hash functions, most notable by the Rabin fingerprinting algorithm.

According to [2, p. 11] this technique has been first used in this context in the "Low Bandwidth File System" (LBFS) and has been subject to a lot of research which proposes refinements for different use cases and in order to overcome some shortcomings shown below. Rabin fingerprinting uses a sliding window which represents the actual hash. By moving the fixed-size window along the data a new byte sequence enters the fingerprint, while the last byte sequence leaves the window. Properties of rolling hashes allow to only perform those two operations on the existing hash and therefore are very efficient in this regard. This is shown exemplarily in Figure 2.1. At the beginning the hash of the whole window of size eight is calculated. By sliding the window across the data the letter "A" leaves the window while "/" enters. The new hash can now be calculated by referring to the old hash, the removed and the added letter.

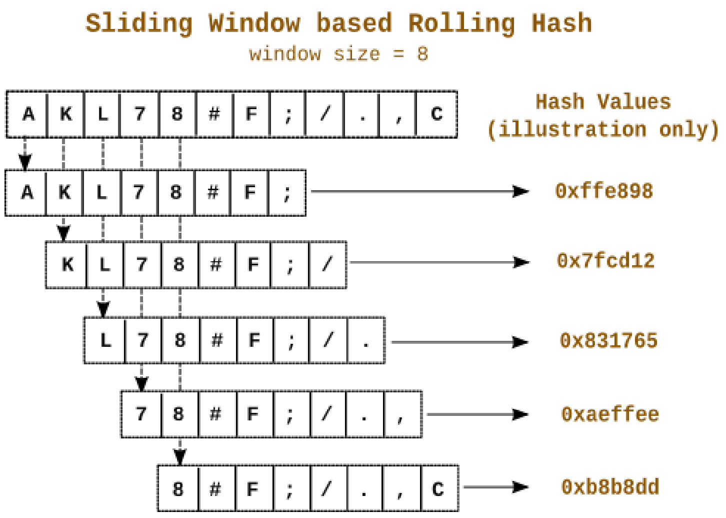


Figure 2.1: Rolling hash example [3]

Whenever a specific condition is detected a cut is made. For example the authors of the LBFS filesystem note in [4] that a chunk is created whenever “the low-order 13 bits of the fingerprint equal a constant value”.

Because of the desired properties of hash functions in general and the details outlined by the author of the Rabin algorithm [5] a uniformed distribution of this hash function is expected. Therefore very large chunks, as well as, small chunks are expected, while the average chunk size can be controlled by a parameter. Like noted in [6] very small chunks increase overhead in metadata handling and large chunks lead to a smaller deduplication ratio.

In practice, artificial cutoff points are assigned with the intention to limit the minimal and maximal size of the chunks with the reasoning noted above. The authors in [7] also note that those cutoff points “are not robust and not reproducible in the case of relatively small inserts or deletes”. Therefore several techniques exist that try to improve the performance of dynamic deduplication in this regard.

In [7] a method is shown which tries to create more segments with a uniform size. This is achieved by increasing the probability of creating a cutoff point with each byte read since the previous chunk and therefore removing the need for artificial cutoff points.

The challenge with very large chunks is also noted in [8], which additionally emphasises that when the probability of their occurrence is high, the same issues arise that are known from static chunking, namely the boundary shift problem. As a solution to this problem an algorithm called “Elastic Chunking” is proposed, which tries to reduce the number of consecutive chunks of maximal size by increasing the maximal chunk size in those cases. This is done by implementing an adjustment policy which keeps track of previous chunks. First experiments show promising results and further research is being conducted with additional data sets.

Another approach is shown in [9] where static chunking is combined with local file similarity. The authors propose an approach where when a file is changed most changes evolve around a specific part of a file. This holds true for many file formats while exceptions like JPEG-2000[1] and archives often handle file size reduction by themselves. Their implementation computes a file similarity hash using the Rabin algorithm, which enables the deduplication process to find a very similar file on the server. The similarity information itself consists out of the hash and offset information. If a match is found on the server the difference of the hash on the local file in comparison to the remote file is calculated. If those two positions are unequal the file changes can be observed and static chunking is used on this specific range. In their evaluation the authors observed a processing time which matches that of static chunking, while achieving a deduplication ratio comparable to that of dynamic chunking. One downside is that the performance is highly dependent on the data type and the locality of the changes.

As can be seen in this section there are many factors that are relevant when deciding on which chunking strategy one should rely on. Some techniques are highly dependent on the use case and the performance of others depend on the layout of the data. In order to built a broadly usable deduplication system which is unopinionated about the underlying data static chunking is used in the remainder.

2.2 Fingerprinting

In order to identify a unique block a secure hashing function like SHA-1 and SHA-256 is used. Depending on the overall requirements an additional byte-for-byte level comparison is feasible, like shown in the approach implemented in [10]. This might especially be required when using a hashing function where collisions have been created like in [11]. Newer implementations might therefore prefer SHA-256 and according to the documentation ZFS even supports SHA-512 [12].

Another aspect of fingerprinting is the handling of storage requirements. As shown in [13] a naive implementation of deduplication would require for 1 PB of data, a block size of 8 B and SHA-128 (160 bit) about 2.5 TB of fingerprinting data. Therefore this approach is not feasible for large datasets. In comparison using SHA-256 and additional metadata ZFS allocates about 320 B for each entry in the deduplication table (DDT) [14]. Therefore the deduplication table would need to reside on disk and every I/O operation would involve additional random access and therefore slow down the file system.

In [15] the author shows an alternative approach and introduces a ZFS prototype where if the DDT does not fit into the memory entries which are not referenced by another chunk are evicted from memory. This technique works as long as the memory is not filled with fingerprints of chunks which are referenced more than once.

A few more basic concepts that are used in practice are shown below while following the

[1]`https://github.com/uclouvain/openjpeg/wiki/DocJ2KCodec`

notation introduced in [13]. Among helping with size management of the fingerprinting index those changes can help improving the I/O performance. Due to the high probability of scattering a file across several regions on the storage layer especially the read performance can be improved. This is particularly relevant for appliances outside of backup systems.

2.2.1 Locality-Based

Those systems benefit from the similarity of the alignment of chunks across different files. In general the claim is, that when a chunk occurs in some file the probability of the next chunk is related to the occurrence in another and therefore similar file.

An example implementation is given in [16], where the authors introduce "POD: Performance Oriented I/O". Their goal is to balance the performance of write and read operations of small and large files by dynamically changing the memory utilization of the fingerprinting index with respect to the read cache. An important component of this system is the decision on which chunks actually need to be indexed. This is done by classifying the data into certain categories. Depending on whether a redundancy exceeds a threshold deduplication is performed. This also helps to improve the read performance, as more continuous chunks can be read.

2.2.2 Similarity-Based

This technique can be used where files do not share locality of individual chunks but an overall similarity.

In [17] "Extreme Binning" is introduced. The intention is to store a similarity hash of every file in memory. This hash can then be used to read the actual chunk hashes from secondary storage in one consecutive I/O operation. The key component here is the calculation of the similarity hash, which is handled by the Broder's theorem. In this context the theorem states that if two files have similar chunks the probability that both result in the same hash is high.

2.2.3 Flash-Assisted

With increased IOPS capabilities in recent storage technologies the fingerprinting data does not have to reside in the memory at all costs. With NVMe drives achieving over 1 millions random reads per second alternatives emerge. One example is shown in [18] where, among other optimizations, a key-value store is used to handle the fingerprints on flash drives. Modern key-value stores like RocksDB[2] have native support for those appliances nowadays.

[2]https://rocksdb.org

2.3 Use Cases

In this section use cases for deduplication are shown. The most common use case for this feature is when backups are involved. However, deduplication can be used in all sorts of scenarios and in a variety of environments.

2.3.1 Backups

Backups are ideal for deduplication as in many cases the majority of the data stays the same in between backup runs. In those cases e.g. file-level deduplication is a fast and resource efficient way to achieve a worthwhile deduplication ratio. Another benefit is that when deduplication is performed on the client only chunks that are new or have changed have to be transmitted across a potentially slow network connection.

2.3.2 Container images

In OS images a lot of data is redundant. According to Microsoft virtualization related data can be reduced by 80-90%[3] when using deduplication. In [19] researchers have built a tool called Slimmer that is capable of applying deduplication on Docker registries. They found out that only 3% of the data is unique and when using file-level deduplication, paired with compression, storage savings up to a factor of 24.8 are possible.

2.3.3 HPC

In [20] a study is performed on datasets residing on HPC storage systems. It was shown that 20-30% of the data can be saved when deduplication is being used. In practice it is still questionable on how well deduplication can be used on a productive system due to the large overhead when working on potentially thousands of petabytes of data and the existing I/O paths and distributed environments.

However, the study also shows that it might be possible to derive workflow improvements from analysing the individual project spaces. In some cases exact file copies were found and unpacked archives were kept. Instead of enforcing deduplication across the whole file system it therefore might be possible to increase user awareness of their individual impact on the available storage in the HPC facility.

[3] `https://docs.microsoft.com/en-us/windows-server/storage/data-deduplication/overview`

3 Related Work

3.1 ZFS

ZFS is a well-known, modern and broadly used file system, which also includes a volume manager. It provides state of the art features like software defined RAID support, checksums, snapshots, replication, compression and deduplication. Development of this file system started in 2001 during Sun Microsystems ownership and is nowadays fully open source and managed under the name OpenZFS[1].

At its foundation ZFS uses blocks of configurable size which are handled according to a copy-on-write mechanism. All modifications therefore result in a new block being written. Whenever a block is read the content is verified based on a checksum calculated by either Fletcher-2, Fletcher-4 or SHA-256 during the write process. The requirements for deduplication are therefore deeply integrated into the roots of the filesystem.

A deduplication specific feature in ZFS is the deduplication table, which has been introduced in Section 2.2. In order to estimate if deduplication provides a benefit for the user-specific use case the `zdb` utility can be used. An example can be seen in Listing 3.1. The "refcnt" column denotes how many times a block is referenced. This is then shown from an allocation and from a reference perspective. A single block with a logical size of 128 kB is referenced over 2 million times resulting in a theoretical physical size of 272 GB. However, when using deduplication this block is stored only once, which is shown in the "allocated" group. This leads to a physical deduplication ratio of $\frac{\text{referenced PSIZE}}{\text{allocated PSIZE}} = \frac{306\,\text{GB}}{33.3\,\text{GB}} = 9.18$ and the actual use of deduplication should be considered in this case.

[1]https://openzfs.org/wiki/Main_Page

```
user@host:~$ zdb -S <POOLNAME>
Simulated DDT histogram:

bucket              allocated                       referenced
------   ----------------------------   ----------------------------
refcnt   blocks   LSIZE   PSIZE   DSIZE   blocks   LSIZE   PSIZE   DSIZE
------   ------   -----   -----   -----   ------   -----   -----   -----
     1     555K   69.3G   33.3G   33.3G     555K   69.3G   33.3G   33.3G
    2M        1    128K    128K    128K    2.13M    272G    272G    272G
 Total     555K   69.3G   33.3G   33.3G    2.67M    342G    306G    306G

dedup = 9.18, compress = 1.12, copies = 1.00,
dedup * compress / copies = 10.26
```

Listing 3.1: Evaluation of deduplication before actual activation [21]

3.2 OpenDedup

OpenDedup[2] provides deduplication features as a POSIX compliant file system and can serve as an intermediate layer between local storage and cloud storage providers. It is mainly intended for backup purposes and is compatible with commonly used applications for similar use cases like Backblaze and NetBackup.

In contrast to the deduplication technique used in ZFS, OpenDedup provides static chunking with block sizes set in the range of 4 kB to 128 kB and dynamic chunking using Rabin fingerprinting. The challenge of finding optimal chunk sizes can also be seen here as the minimal and maximal size of those chunks are predefined and set to 4 kB and 32 kB, respectively.

When storing the data on remote storage a local cache is used for quicker access to the most recent data. The remote storage also serves as an operator for data resilience purposes, as for example AWS S3 is capable of handling data loss in at least two facilities[3].

[2]https://opendedup.org
[3]https://docs.aws.amazon.com/AmazonS3/latest/userguide/DataDurability.html

4 JULEA

In this chapter a deduplication implementation for JULEA is introduced. First of all a minimal viable implementation in regard to possible variants described in Chapter 2 is highlighted. Further on the general architecture of JULEA and the integration of this new feature is shown. Finally, several scenarios are evaluated. The source code is available on Github[1].

4.1 Introduction

JULEA is a storage framework developed by Michael Kuhn and has been first introduced in [22]. It provides several backends that can be used for data and metadata purposes, which can either run on the client or the server side. Those backends are implemented in user space and restrictions exposed e.g. by the VFS layer do not apply.

The point where JULEA intervenes is shown in Figure 4.1. On the left side a traditional, rigid HPC stack can be seen. An application uses a scientific I/O library, like NetCDF, that depends on other frameworks by itself. This is motivated by the need for an easy and in this case domain specific access to I/O functionalities. Those requests are then passed along other libraries like HDF5 and finally MPI-IO is used for access to a parallel, distributed file system like Lustre. As in general the ease of use declines as the potential for new concepts rises throughout this stack it becomes a great challenge to implement new features.

On the right the proposed stack introduced by JULEA can be seen. In this case JULEA is used as a transparent interface between previous layers and allows for easier extensions on the I/O path.

This enables researchers to implement various features more easily and the possibility to rely on modern technology. As those restrictions do not apply data backends with support for high-level APIs like RADOS[2] and GIO[3] exist. In order to handle metadata in various ways support for transactional databases like MySQL and SQLite were implemented, as well as, key-value stores like LMDB and LevelDB.

Those backends share an API and new variants have to implement either the data or

[1]`https://github.com/Julius-Plehn/julea/tree/feature/dedup`
[2]`https://docs.ceph.com/en/latest/rados/api/librados-intro/`
[3]`https://developer.gnome.org/gio/stable/`

the metadata API. Deduplication relies heavily on handling of small I/O operations as well as the availability of efficient data structures for metadata. Those prerequisites are fulfilled by JULEA and in the next sections additional JULEA concepts and the implementation details are shown.

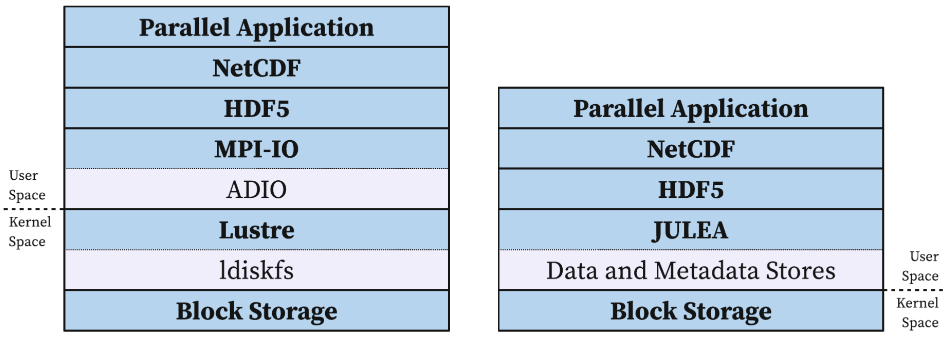


Figure 4.1: JULEA Stack [22]

4.2 Concepts

Before getting into the implementation details of the new deduplication feature, further JULEA specific concepts are introduced. This is done by highlighting how a similar feature, the regular `JItem` client, is built and what components are involved when performing a write or a read operation.

First of all, some important components that are used throughout JULEA are introduced.

Batches

In general all operations are performed in batches, following a specific semantics. This correlates to the term “transactions” used in databases and even the integrated semantics behave similar to the popular ACID properties. Several available semantics can be integrated into a template and a new batch can be created from it. Among others an `atomicity` semantic enables JULEA to specify when intermediate changes introduced by an operation are available to other clients. The `consistency` semantic can be used to specify when modifications will be applied to the backend and the `ordering` semantic can be used to optimize the order of operations within a batch. Combining those constraints it is possible to emulate POSIX semantics. Another purpose of batches is that they are used to aggregate several operations for performance optimization purposes.

Collections

As the item client mimics behavior found in cloud environments, specifically object storage related APIs like S3, collections are used to separate items logically in a flat hierarchy. The items themself are identified by a name/key attribute and both identifiers can be iterated. Depending on used security semantics those entities are only available to the specific user or group.

Objects

Objects are used to store the actual data and they come with two different implementations. The first one uses a distribution to split the object onto several servers, while the second one hashes the objects name in order to assign a single object in its entirety to a single server. As of now the user can either choose to distribute the data according to the round-robin algorithm or according to weights. It is also possible to assign a single server. In this case a random server is chosen when the distribution is created.

The existing `JItem` client uses the distributing variant, while the new deduplication capable client uses the regular object store. When using the regular client the distributed object is initialized when the item is created. At this point the object is assigned a namespace and the file path is used as a name. Furthermore the previously specified distribution is attached. This part only serves as an initialization for further I/O operations and the object is created in the batch which is used for the creation of the item itself.

Behind the scenes and independent from the object variant used, the I/O is passed along to the configured `JBackend` implementation which implements the `object` interface. This works similar for other backends like databases and key-value stores.

4.3 Implementation

Based on the findings in Chapter 2 a deduplication system with the following characteristics has been developed. The deduplication uses static chunking in an online manner. Therefore all chunks are assigned at immediate run-time and only new, unknown chunks are written. Garbage collection is performed at the same time as whenever a chunk is not referenced anymore a cleanup operation is performed. To provide the best possible resistance to hash collisions SHA-256 is used.

Deduplication is implemented as a new item client, similar to the existing `JItem` client and therefore can be used directly by other applications. How this benefits our use case can be seen in Figure 4.2. This new client runs on the compute nodes and only unique and unseen chunks have to be sent across the TCP network. This is especially helpful when the deduplication ratio (DR) is high or the network is slow.

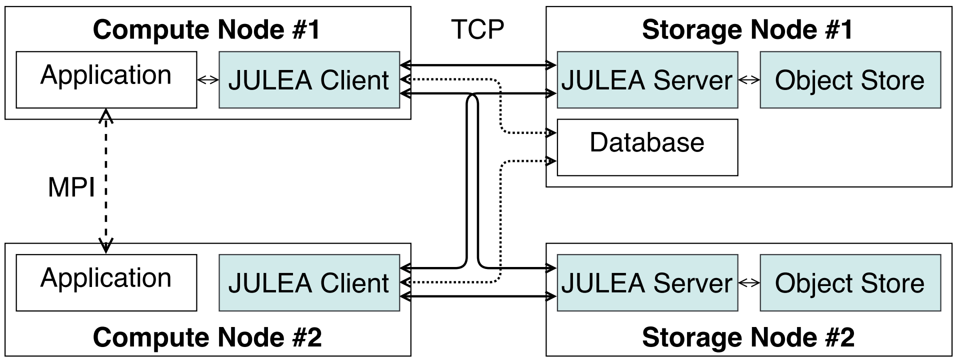


Figure 4.2: JULEA Client-Server Architecture [23]

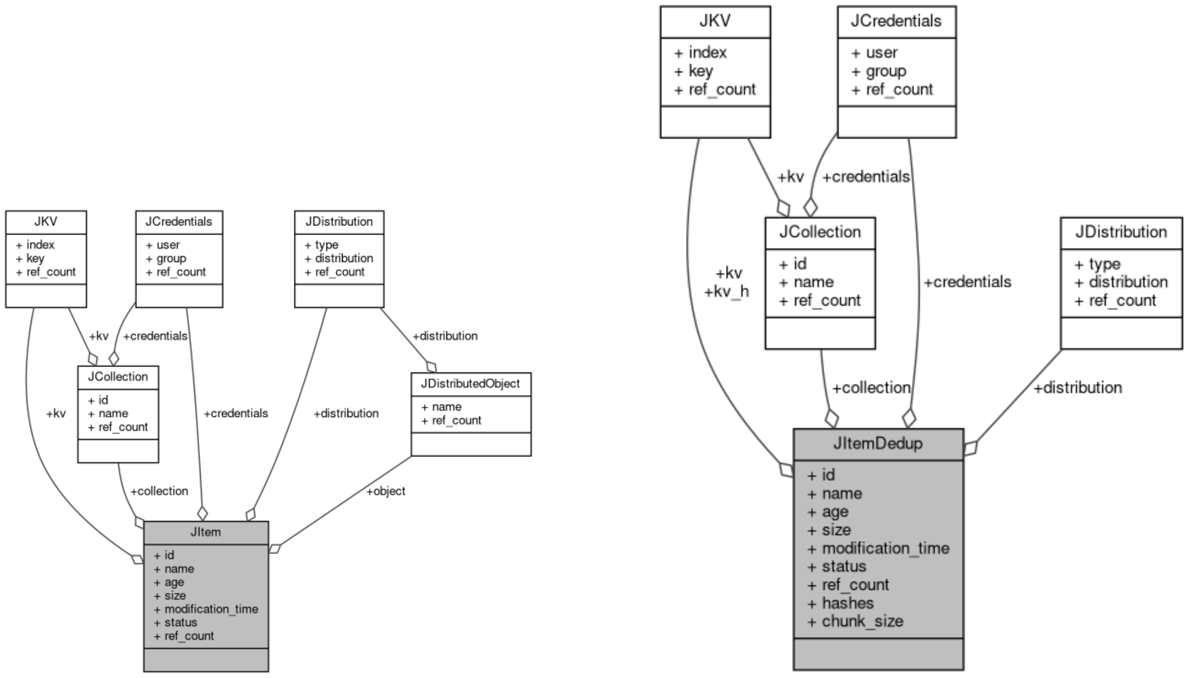
This new deduplication capable client is called `JItemDedup`. Its interface closely resembles that of the `JItem` client and shares many functions which are prefixed by `j_item_dedup_-` instead of `j_item_`. Most notable, from a user perspective, the read and write functions take the same parameters and only an additional and in many situations optional function `j_item_dedup_set_chunk_size` can be used to set an individual chunk size for a specific item.

In the next sections the overall structure of the new client is introduced. Additionally, several important I/O operations are highlighted.

4.3.1 JItemDedup

The `struct` of the new client shares many fields of the previous client. The biggest difference is that no object entity for actual data storage is attached to the new client. Previously the data was attached to a single distributed `JDistributedObject`. However, as the data is chunked according to a policy, the distribution happens as new data arrives and a chunk is represented by an independent `JObject`. This is required as a single object can be shared among many deduplicated items and the lifecycle of those objects is linked to its reference counter that is shared globally.

In order to handle the hashes of the item a new key-value member in the `struct` is used (`kv_h`). This is an addition to the existing `kv` member, which is still used to handle metadata of the item itself, for example the name and the modification time. The content of the key-value store is serialized/deserialized depending on the lifecycle of the item and stored in `GArray *hashes`. Another new member is called `chunk_size` which stores the size in which the chunks are going to be split into. These changes and additions can also be seen in Figure 4.3.



(a) JItem Struct (b) JItemDedup Struct

Figure 4.3: Differences between JItem and JItemDedup

4.3.2 Operations

In this section details about read and write operations are shown. This is done by outlining which influence the state of the overall system has on the operations, especially on the write path. This is of special interest when working on chunked and distributed data. When performing read or write operations a pointer to a buffer for holding the data is given, as well as, the data length and an offset within the data. It is therefore possible to read and write a specific part of an item and the deduplication client has to be able to cope with those cases.

Independent of the type of I/O some chunk related calculations have to be made. This includes the identification of the very first and last chunk, the total number of chunks and the number of bytes which have to be processed in the last chunk.

The write operation is also shown in Listing 4.1 which helps to relate the overall steps needed to perform deduplication with the actual code.

Write: New item

In this situation a new item is created and no previous data has been written to this item. Given the size of the data and the defined chunk size the number of total chunks required can be calculated. For every chunk an independent hash has to be calculated. This is shown in line 23-34. In many cases hashing algorithms propose an API which evolves around a context. It is therefore easily possible to add input to the hashing algorithm and to get the final hash which is shown in line 34.

Another important aspect is the calculation of the `len` variable, which denotes how much data has to be written in this particular chunk. The length depends on whether the specific chunk is positioned at the beginning, the end or whether both conditions are fulfilled. Additionally, this variable is influenced by the possibility of the use of an offset. The chunks have to be aligned accordingly and the variable `data_offset` (line 1) keeps track of where the input data has to be read from. The progress is therefore tracked in line 83.

When this is done the key-value store is used to get the total number of references to this chunk by querying the hash. This is shown in line 36-38 where the `chunk_refs` namespace is used. When the callback is executed the variable `refcount` holds the number of references to this specific chunk.

In the best case scenario the hash, respectively the chunk itself, is known to the system. When the hash is unknown a new object has to be created (line 40-58). While previously, in the non-deduplication capable client, a single item consisted out of a single distributed object, a deduplicated item consists out of many, independent objects. Therefore, when a new object is created a separated namespace ("chunks") is used and the object itself is named after its hash (line 43). The write operation on the object is similar to the way the hash has been calculated, except that now an additional offset for the object has to be used.

A direct connection between the independent objects and the deduplicated item does not exist. The final step it to increment the reference counter of the new object in the key-value store (line 60-70).

Finally, the hash of the chunk is added in order of their occurrence (line 82).

Write: Existing item

In this case an item already exists and data is written using an offset. While in the regular item the data is simply written at that specified offset some extra steps are necessary when using deduplication.

As this deduplication variant uses a static chunk size the data has to be hashed accordingly. When using an offset it is now required to recreate the existing chunk and change it so that the new data, that is included in that specific part, forms a new chunk and a new hash can be calculated - including the previous part. This can be seen in line 25 and 32. In both cases a previously created buffer is used and fitted according to the layout of the new data.

In order to write the actual new data some old data has to be read first. It is therefore required to calculate the offset within the chunk with respect to the global offset passed to the write function. The next step is to read this specific part using the associated object, which is identified by the used hash at this position. The required hash is easily available as the hashes are deserialized into a `GArray` and they are ordered according to the occurrence of their associated chunks.

The remaining steps are similar to the ones shown in the previous scenario with one important exception. It is possible that some old chunks have been modified. However, this is not reflected in the current state because when a chunk at a certain position has changed a new object is either created or a reference counter has been incremented. After the most recent hashes have been serialized and added to the current item a garbage collection step is required. This means that if a chunk is processed the final step is to check if previously the same chunk ocurred at the same position. If this is not the case the associated reference counter is decremented and when no further references are needed the chunk is deleted. This can be seen in line 72-81 where the hash is removed from the `GArray`, which is present within the item. The actual garbage collection step is performed within the `j_item_unref_chunk` function.

```
1  guint64 data_offset = 0;
2  for (guint64 chunk = 0; chunk < chunks; chunk++)
3  {
4    gchar* hash;
5    guint32 refcount = 0;
6    guint64 len = item->chunk_size;
7
8    algo_array[hash_choice]->init(hash_context);
9
10   if (chunk == 0 && chunk == chunks - 1)
11   {
12     len = item->chunk_size - chunk_offset - remaining;
13   }
14   else if (chunk == 0)
15   {
16     len = item->chunk_size - chunk_offset;
17   }
18   else if (chunk == chunks - 1)
19   {
20     len = item->chunk_size - remaining;
21   }
22
23   if (chunk == 0)
24   {
25     algo_array[hash_choice]->update(hash_context, first_buf,
        ↪  chunk_offset);
26   }
27
```

```c
  algo_array[hash_choice]->update(hash_context, (const gchar*)data + data_offset, len);

  if (chunk == chunks - 1)
  {
    algo_array[hash_choice]->update(hash_context, last_buf, remaining);
  }
  algo_array[hash_choice]->finalize(hash_context, &hash);

  chunk_kv = j_kv_new("chunk_refs", (const gchar*)hash);
  j_kv_get_callback(chunk_kv, j_item_hash_ref_callback, &refcount, sub_batch);
  ret = j_batch_execute(sub_batch);

  if (refcount == 0)
  {
    guint64 extra_offset = 0;
    chunk_obj = j_object_new("chunks", (const gchar*)hash);
    j_object_create(chunk_obj, batch);

    if (chunk == 0 && chunk_offset > 0)
    {
      j_object_write(chunk_obj, first_buf, chunk_offset, 0, bytes_written, batch);
      extra_offset = chunk_offset;
    }

    j_object_write(chunk_obj, (const gchar*)data + data_offset, len, extra_offset, bytes_written, batch);

    if (chunk == chunks - 1 && remaining > 0)
    {
      j_object_write(chunk_obj, last_buf, remaining, item->chunk_size - remaining, bytes_written, batch);
    }
  }

  new_ref_bson = bson_new();
  bson_append_int32(new_ref_bson, "ref", -1, refcount + 1);

  {
    gpointer value;
    guint32 value_len;
    value = bson_destroy_with_steal(new_ref_bson, TRUE, &value_len);

    j_kv_put(chunk_kv, value, value_len, bson_free, sub_batch);
  }
  ret = j_batch_execute(sub_batch);

  if (chunk < old_chunks)
  {
    gchar* old_hash = g_array_index(item->hashes, gchar*, first_chunk +
     ↪  chunk);
    if (g_strcmp0(old_hash, (gchar*)hash) != 0)
    {
      j_item_unref_chunk(old_hash, batch);
      g_array_remove_index(item->hashes, first_chunk + chunk);
      g_free(old_hash);
    }
  }
  g_array_insert_val(item->hashes, first_chunk + chunk, hash);
  data_offset += len;
}
algo_array[hash_choice]->destroy(hash_context);
```

Listing 4.1: Write process (Shortened)

Read

As mentioned previously the read process requires similar precalculations like the write path. Depending on the given offset the required number of chunks is calculated. Additionally, depending on the position of the individual chunk, the length that has to be read from the referenced object is chosen.

The final step is to iterate throughout the required objects, which are referenced by their hashes and to recreate the requested item by appending the data to an allocated buffer.

5 Evaluation

In this chapter several scenarios are evaluated. In the first part the deduplication performance on individual files is shown. Finally, a specific use case is highlighted. For evaluation purposes three different datasets are analysed. One dataset consists out of linux kernel archives[1], either used as individual files or as an archive. Another dataset was taken from a previous ECHAM[2] run and is stored as a NetCDF file. Finally, DEDISbench[3] was used to generate realistic datasets based on collected metrics.

5.1 Single File Performance

In the first test several linux kernels have been combined into a single tar-archive resulting in a combined size of 9.8 GB. Compression was explicitly disabled. The influence of different chunk sizes and their impact on the runtime can be seen in Figure 5.1. It is expected that the deduplication rate increases as the chunk size decreases and this behavior can be seen here. A chunk that is 4096 B large and introduces a specific deduplication rate has to, by definition, achieve the same or a better deduplication rate when the chunk size becomes half of the previous size.

This file format also serves as an example on what influence the packaging and the internal layout of the individual data chunks might have on the optimal chunk size. As noted in the tar documentation[4] data is stored in blocks, which are 512 B large and several blocks are read and written at once in a unit called “record”. Furthermore, each record is separated by gaps and the occurrence of those therefore depend on the number of blocks in a single record. It is now either up to the user to define the correct chunk size when using deduplication or up to the deduplication policy to decide on the individual chunk size by using dynamic chunking, which has been introduced earlier.

For reference another file has been benchmarked, which can be seen in Figure 5.3. The results are very similar, however in this case the achievable maximal deduplication rate is far less than in the previous example. The performance in general is heavily influenced by the actual data that is written. Another aspect is the layout of the data. The internal mechanism on how to save data in either a tar-archive or in a NetCDF file varies vastly.

[1]`https://github.com/torvalds/linux/releases`

[2]`https://mpimet.mpg.de/en/science/models/mpi-esm/echam/`

[3]`https://github.com/jtpaulo/dedisbench`

[4]`https://www.gnu.org/software/tar/manual/html_node/Blocking-Factor.html#Blocking-Factor`

Even in a scenario where exactly the same data is written the deduplication ratio could be completely different and could also depend on a different chunk size.

Another aspect is the influence on the runtime, which increases as the chunk size decreases. This relationship can also be seen in Figure 5.2, where in a benchmark scenario a large portion of the time was spent on metadata operations. Those type of operations increase when more chunks have to be processed. The measurements were made using the perf utility[5]. Perf is useful to gather insights into the internal behavior of an application and the associated kernel by capturing events that are emitted by the operating system. In this case the `cycles` event was captured meaning that by counting the CPU cycles a specific part of an application took it is possible to tell where most time was spend within the execution. In this case it can be observed that within `j_item_dedup_write` about 11.6 % was spent on operation related to the key-value store. This is unique to the deduplication feature as batches were executed within the write operation itself.

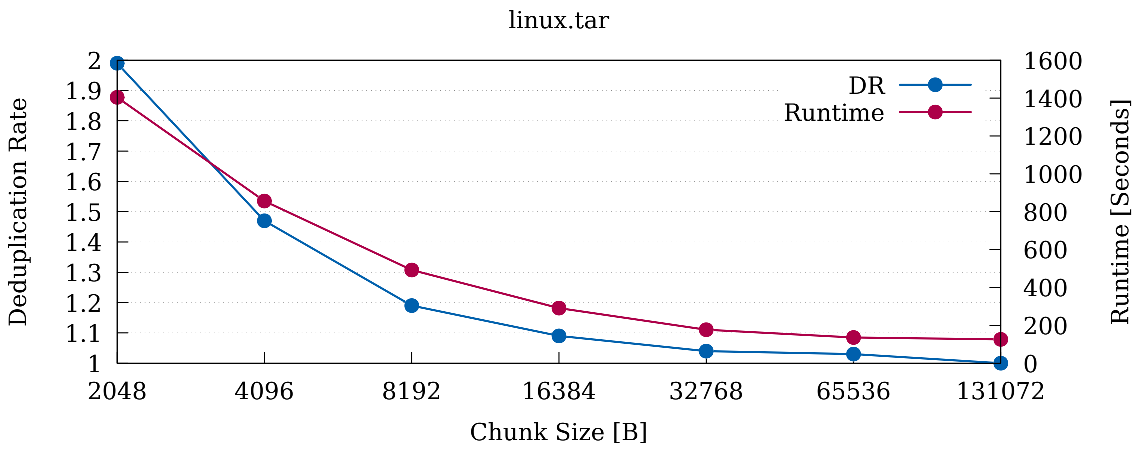
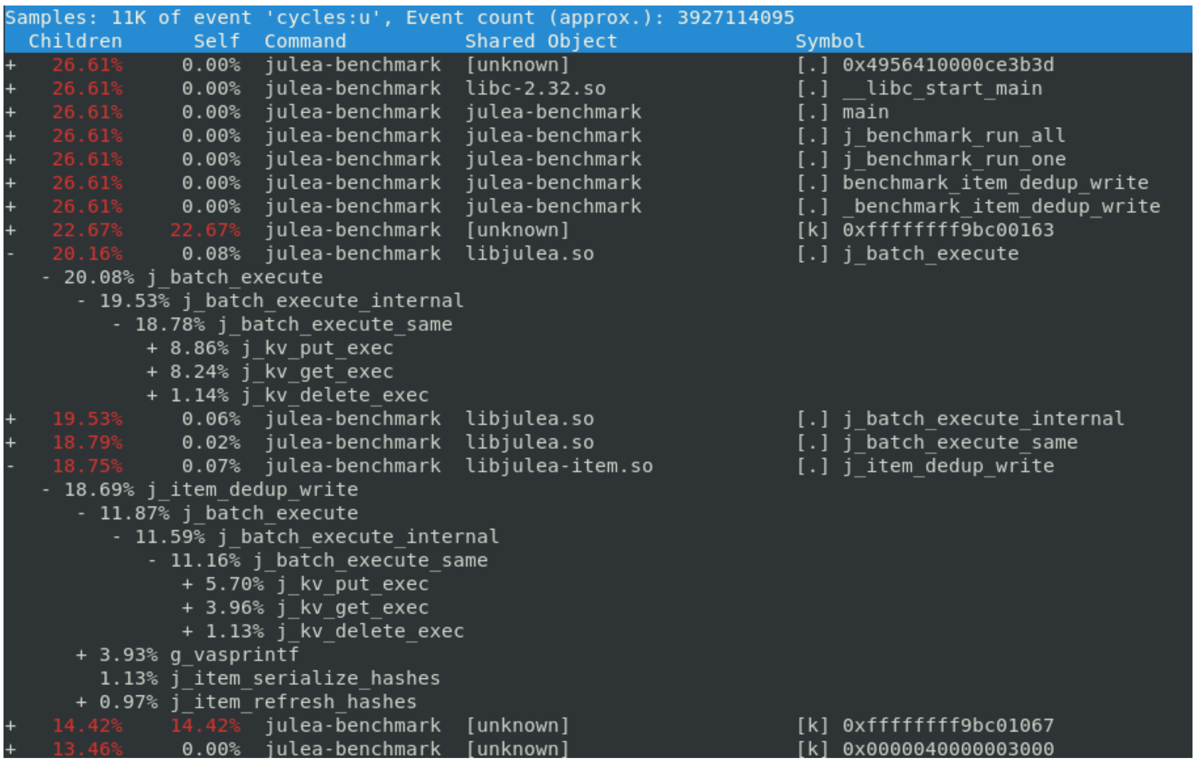


Figure 5.1: Deduplication performance on tar-archive

[5]`https://perf.wiki.kernel.org/index.php/Main_Page`



```
Samples: 11K of event 'cycles:u', Event count (approx.): 3927114095
  Children      Self  Command          Shared Object                 Symbol
+   26.61%     0.00%  julea-benchmark  [unknown]                     [.] 0x4956410000ce3b3d
+   26.61%     0.00%  julea-benchmark  libc-2.32.so                  [.] __libc_start_main
+   26.61%     0.00%  julea-benchmark  julea-benchmark               [.] main
+   26.61%     0.00%  julea-benchmark  julea-benchmark               [.] j_benchmark_run_all
+   26.61%     0.00%  julea-benchmark  julea-benchmark               [.] j_benchmark_run_one
+   26.61%     0.00%  julea-benchmark  julea-benchmark               [.] benchmark_item_dedup_write
+   26.61%     0.00%  julea-benchmark  julea-benchmark               [.] _benchmark_item_dedup_write
+   22.67%    22.67%  julea-benchmark  [unknown]                     [k] 0xffffffff9bc00163
-   20.16%     0.08%  julea-benchmark  libjulea.so                   [.] j_batch_execute
   - 20.08% j_batch_execute
      - 19.53% j_batch_execute_internal
         - 18.78% j_batch_execute_same
            + 8.86% j_kv_put_exec
            + 8.24% j_kv_get_exec
            + 1.14% j_kv_delete_exec
+   19.53%     0.06%  julea-benchmark  libjulea.so                   [.] j_batch_execute_internal
+   18.79%     0.02%  julea-benchmark  libjulea.so                   [.] j_batch_execute_same
-   18.75%     0.07%  julea-benchmark  libjulea-item.so              [.] j_item_dedup_write
   - 18.69% j_item_dedup_write
      - 11.87% j_batch_execute
         - 11.59% j_batch_execute_internal
            - 11.16% j_batch_execute_same
               + 5.70% j_kv_put_exec
               + 3.96% j_kv_get_exec
               + 1.13% j_kv_delete_exec
      + 3.93% g_vasprintf
        1.13% j_item_serialize_hashes
      + 0.97% j_item_refresh_hashes
+   14.42%    14.42%  julea-benchmark  [unknown]                     [k] 0xffffffff9bc01067
+   13.46%     0.00%  julea-benchmark  [unknown]                     [k] 0x0000040000003000
```

Figure 5.2: Impact of metadata operations

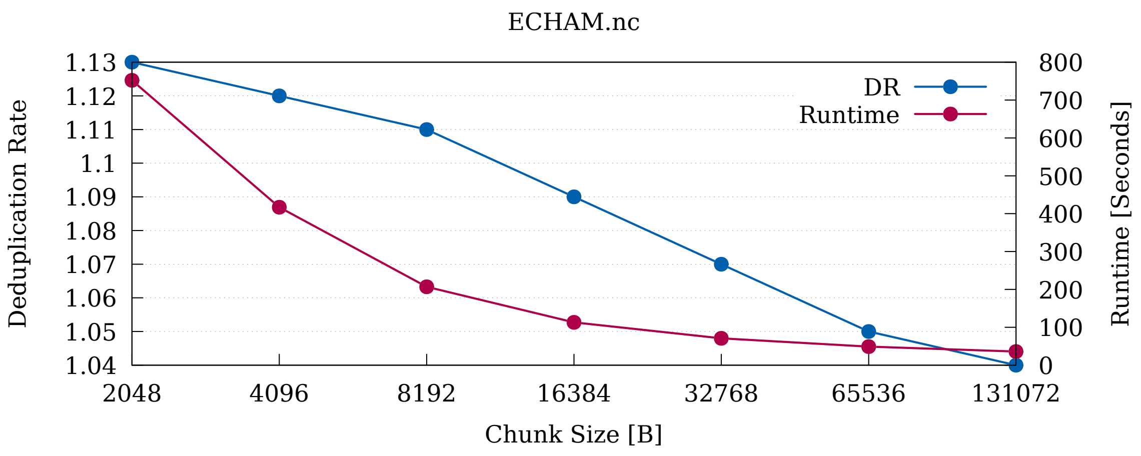


Figure 5.3: Deduplication performance on ECHAM NetCDF output

The following two plots are based on data generated by the DEDISbench tool. This tool can be used to simulate data that is derived of an input file that represents a distribution of blocks and their probability of occurrence. An example can be seen in the Github repository[6]. Each line represents a number of blocks and how many duplicates should be

[6]`https://github.com/jtpaulo/dedisbench/blob/master/conf/dist_archival`

created accordingly. This project provides several distributions by default and two have been used for analysis. However is is also possible to create custom distributions based on own data using the DEDISgen utility. An important characteristic when creating those files is the selection of the block size. This also highlights the challenge introduced by the use of static chunking in comparison to the use of dynamic chunk sizes.

As can be seen in Figure 5.4 a distribution according to archival data and kernel data has been benchmarked. The analysed file has been created using a block size of 128 000 B. While the given deduplication rate is barely over one using the same block size, the deduplication rate increases drastically when reducing the static chunk size. Similar to the performance seen in Section 5.2 the storage of kernels is an ideal use case for deduplication and a good example for the usage within backup or versioning use cases in general.

Another test was made using an unaligned chunk size of 128 123 B. As expected the achieved deduplication rate was one and therefore not a single block has been reduced.

In Figure 5.5 the amount of created chunks in relation to the chunk size is shown. The benchmark scenario and the used block size is the same as previously. It can be seen that within a deduplicatable chunk the data stays the same for a whole block. Therefore a written block of size 128 000 B can be chunked into smaller chunks and the number of chunks needed stays the same. However, when the chunk size is misaligned every chunk is unique and the deduplication ratio is one, as shown in the previous figure. This also results in a huge amount of required chunks as each one is unique. When using a chunk size of 256 000 B deduplication is possible to a certain extend and the data can be stored by using fewer chunks.

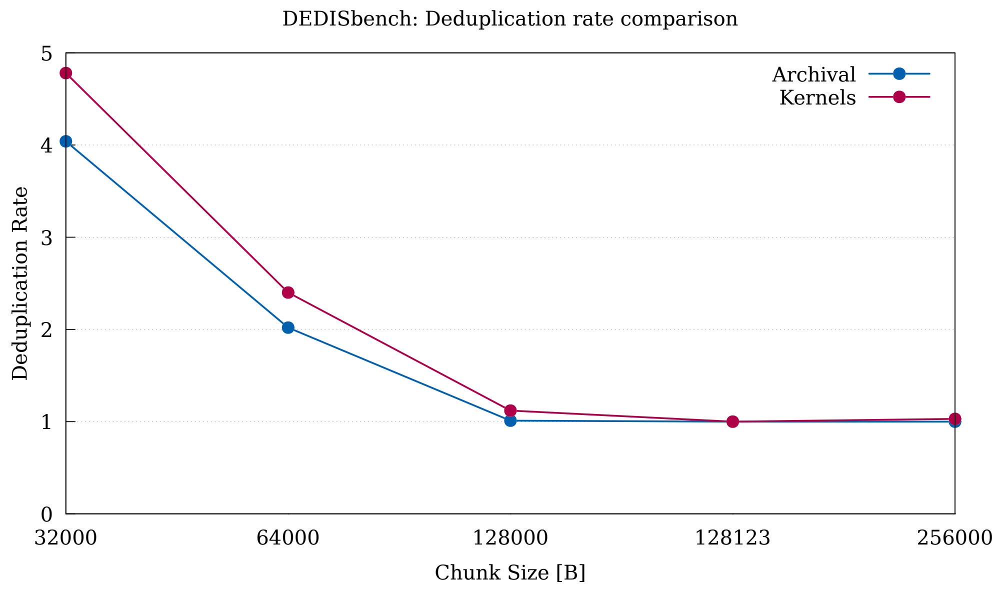


Figure 5.4: DEDISbench with misaligned chunk size

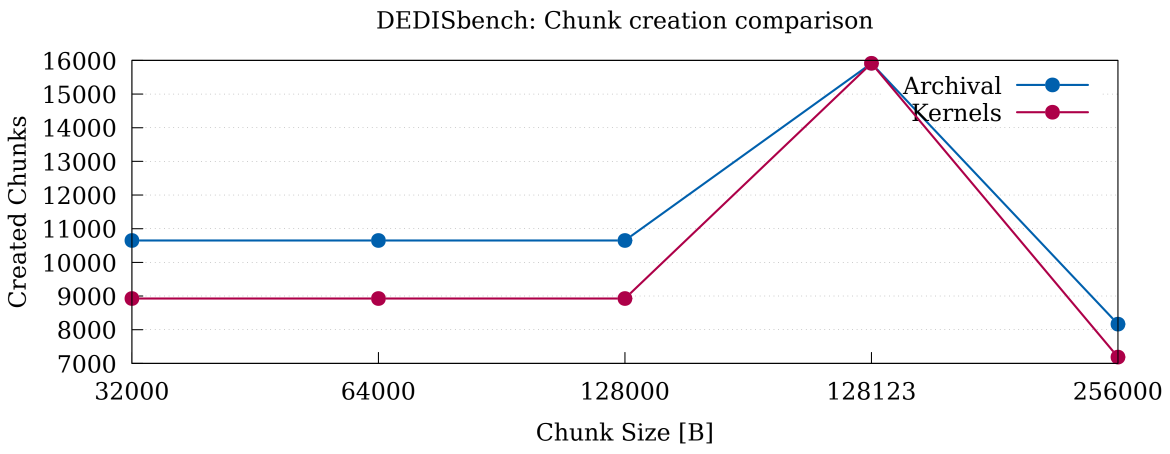


Figure 5.5: Influence of chunk size on metadata generation

5.2 Backup/Versioning Performance

In this section the backup use case is demonstrated. This is done by adding one linux kernel version at a time and tracking the total storage usage of JULEA in comparison to the storage space required on a regular file system without deduplication.

The results can be seen in Figure 5.6, where the blue line represents the storage required within JULEA with deduplication enabled. In comparison the red line represents the accumulated storage requirements without deduplication. The influence of different chunk sizes has also been evaluated, however it has shown to be of little use in this case as deduplication based on the overall file content achieved great results.

As can be seen the initial addition of kernel version 5.12 requires the same space in both cases. Beginning with the addition of kernel version 5.13 a major difference can be observed. While without deduplication the storage requirements rise nearly linearly the space used with deduplication rises only by $\sim 30\,\%$. The remaining content of the kernel can therefore be deduplicated. Another interesting observation is that the differences between the first release candidate of the kernel (5.13-rc1) and the final stable version (5.13) are significant and result in an additional storage use of $\sim 3.8\,\%$. The differences between the intermediate versions however are much smaller. In the end the required physical storage when using deduplication is about 1592 MB and the potential storage savings are significant in comparison to the 10 497 MB required without deduplication.

Another experiment has been done in Figure 5.7. In this setup the POSIX storage backend was located on a Raspberry Pi 3 Model B+ which was connected by WiFi. The metadata backend was located on the client. In this case an additional advantage, besides the previously observed storage savings, can be seen. As the amount of data, that has to be transmitted declines with every additional kernel version, significant time savings are possible. When comparing both plots the correlation between the amount of data that has to be transmitted and transmission time becomes obvious.

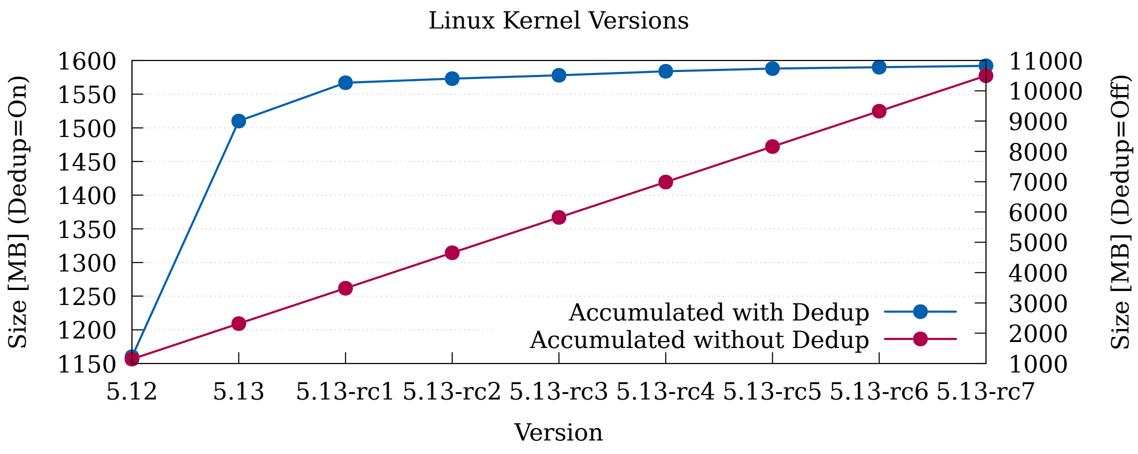


Figure 5.6: Performance on directory containing Linux kernels

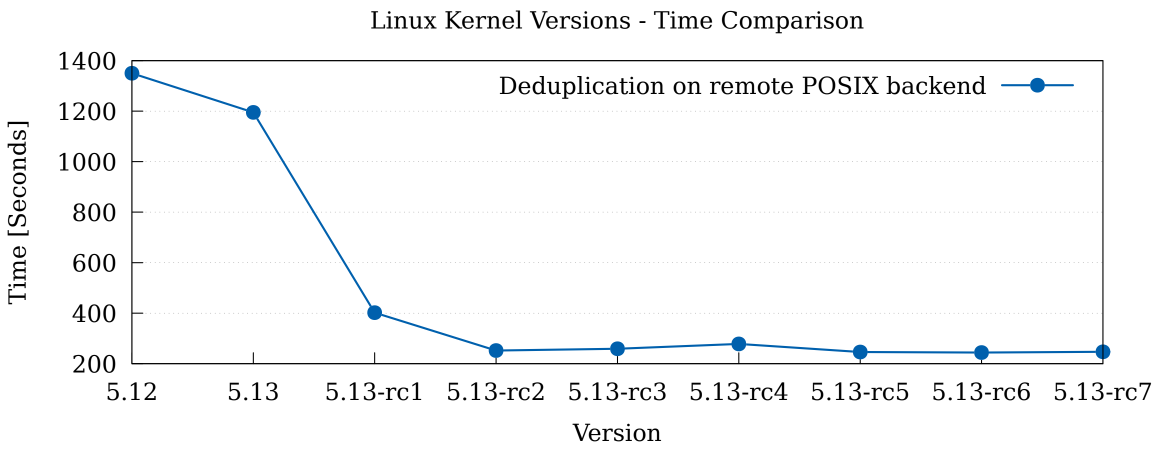


Figure 5.7: Time improvement on directory containing Linux kernels

6 Future Work

The presented deduplication implementation serves as a first usable implementation in order to show the strengths and weaknesses of a basic static chunking policy. In the future two main development targets could be of interest.

First of all, more chunking policies could be explored. The addition of the Rabin fingerprinting algorithm would be a great improvement and could be compared against the performance of the existing one. Additional policies shown in Section 2.2 might also be of interest.

Another step could be to include deduplication in other parts of the JULEA stack. As of now deduplication happens on the client-side and ideally less data needs to be transmitted. Depending on the use case deduplication could happen in the backend directly within the objects. This would also enable experiments using postponed deduplication in which the write process is not slowed down by increased metadata operations and instead, when the available resources allow for it, perform deduplication later on. This also includes garbage collection and metadata heavy tasks.

7 Conclusion

This report has shown that deduplication can achieve significant storage savings, depending on the use case and the handled data, accordingly. In a best-case scenario where multiple iterations of software versions were stored it was possible to save $\sim 70\,\%$ of physical storage when storing the second version indicating a high amount of duplicated data. While this scenario works well in this specific case other scenarios profit from more advanced implementations which do not rely on the static chunking technique. For example it was shown that the influence of a chunk size that is misaligned with the layout of the deduplicatable data has a severe impact on the deduplication rate.

The JULEA framework provides all required functionalities and the necessary expandability in order to integrate those further variants.

Bibliography

[1] E. Manogar and S. Abirami. A study on data deduplication techniques for optimized storage. In *2014 Sixth International Conference on Advanced Computing (ICoAC)*, pages 161–166, 2014. doi:10.1109/ICoAC.2014.7229702.

[2] D. Meister. Advanced data deduplication techniques and their application. 2013.

[3] Moinakg. High performance content defined chunking, 2013. URL: https://moinakg.wordpress.com/tag/rabin-fingerprint/.

[4] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A Low-bandwidth Network File System. pages 174–187, 2001. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.8444.

[5] M.O. Rabin. *Fingerprinting by Random Polynomials*. Center for Research in Computing Technology: Center for Research in Computing Technology. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981. URL: https://books.google.de/books?id=Emu_tgAACAAJ.

[6] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, page 18, USA, 2010. USENIX Association.

[7] Michael Hirsch, Shmuel T. Klein, Dana Shapira, and Yair Toaff. Dynamic determination of variable sizes of chunks in a deduplication system. *Discrete Applied Mathematics*, 274:81–91, 2020. Stringology Algorithms. URL: https://www.sciencedirect.com/science/article/pii/S0166218X18303962, doi:https://doi.org/10.1016/j.dam.2018.07.015.

[8] Wenlong Tian, Ruixuan Li, Zhiyong Xu, and Weijun Xiao. Does the content defined chunking really solve the local boundary shift problem? In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2017. doi:10.1109/PCCC.2017.8280445.

[9] Young Chan Moon, Ho Min Jung, Chuck Yoo, and Young Woong Ko. Data deduplication using dynamic chunking algorithm. In *Proceedings of the 4th International Conference on Computational Collective Intelligence: Technologies and Applications - Volume Part II*, ICCCI'12, page 59–68, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-34707-8_7.

[10] 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*, SYSTOR '09, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1534530.1534539`.

[11] Stevens, Bursztein, et al. Announcing the first SHA1 collision. URL: `https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html`.

[12] Checksums and Their Use in ZFS — OpenZFS documentation. URL: `https://openzfs.github.io/openzfs-docs/Basic%20Concepts/Checksums.html`.

[13] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104:1–30, 09 2016. `doi:10.1109/JPROC.2016.2571298`.

[14] Dominic Kay. How To Size Main Memory for ZFS Deduplication. URL: `https://www.oracle.com/technical-resources/articles/it-infrastructure/admin-o11-113-size-zfs-dedup.html`.

[15] Matt Ahrens. Zero performance overhead OpenZFS dedup, 2019. URL: `https://openzfs.org/w/images/8/8d/ZFS_dedup.pdf`.

[16] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. POD: Performance Oriented I/O Deduplication for Primary Storage Systems in the Cloud. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 767–776, May 2014. `doi:10.1109/IPDPS.2014.84`.

[17] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. 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, Sep. 2009. `doi:10.1109/MASCOT.2009.5366623`.

[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up Inline Storage Deduplication Using Flash Memory. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 16, USA, 2010. USENIX Association.

[19] Nannan Zhao, Vasily Tarasov, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit Warke, Mohamed Mohamed, and Ali Butt. Slimmer: Weight Loss Secrets for Docker Registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 517–519, July 2019. ISSN: 2159-6190. `doi:10.1109/CLOUD.2019.00096`.

[20] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *SC '12: Proceedings of the International Conference on High Performance Computing,*

Networking, Storage and Analysis, pages 1–11, November 2012. ISSN: 2167-4337. doi:10.1109/SC.2012.14.

[21] ZFS auf Linux/ Deduplizierung – Wikibooks, Sammlung freier Lehr-, Sach- und Fachbücher. URL: https://de.wikibooks.org/wiki/ZFS_auf_Linux/_Deduplizierung.

[22] Michael Kuhn. JULEA: A Flexible Storage Framework for HPC. In Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf, editors, *High Performance Computing*, pages 712–723, Cham, 2017. Springer International Publishing.

[23] Kira Duwe and Michael Kuhn. Deliverable D1: Report - Coupled Storage System for Efficient Management of Self-Describing Data Formats (CoSEMoS), 2021. URL: https://parcio.ovgu.de/Research/CoSEMoS.html.

Appendices

List of Figures

List of Listings