Karlsruher Institut für Technologie

Evaluating the Feasibility of Inline FS Deduplication without Main Memory Caches

Studienarbeit
von

Christian Burkert

an der Fakultät für Informatik

Verantwortlicher Betreuer:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dipl.-Inform. Konrad Miller

Bearbeitungszeit: 01. Juni 2012 – 09. November 2012

Deutsche Zusammenfassung

In den letzten Jahren wurde die Vermeidung von Datenduplikaten als spezielle Form der Kompression immer wichtiger. Besonders im Bereich großer gewerblicher Anlagen zur Datensicherung und -archivierung soll Deduplikation den Einsatz von Festplattensystemen attraktiver machen und die etablierten Magnetbandspeicher ablösen. In diesen System werden typischerweise große Tabellen im Hauptspeicher vorgehalten um eine schnelle Erkennung von Duplikaten zu gewährleisten. Entsprechend hoch sind die Ressourcenanforderungen.

Die vorliegende Studienarbeit untersucht, ob durch den Einsatz von Solid-State-Drives (SSD) auf Tabellen und Caches im Hauptspeicher verzichtet werden kann. Dazu wurde das Dateisystem *ext4fs* um eine cachelose Dedupliaktionsfunktion erweitert und getestet. Die Tests ergaben zeitliche Mehrkosten von 7 bis 55 Prozent verglichen zum unveränderten Dateisystem. Basierend auf diesen Ergebnisen scheint in Verzicht auf Caches in Umfeld von optimierten, gewerblichen Speichersystemen auch bei einer zukünftigen Verfügbarkeit von günstigen SSDs nicht sinnvoll. Dagegen könnte im privaten Einsatz der Verzicht vorteilhaft sein, da hier verschiedene Dienste auf einer Hardware integriert sind und durch die Hauptspeichereinsparung andere Dienste profitieren könnten.

Acknowledgments

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Karlsruhe, 9th November 2012

Contents

Chapter 1

Introduction

In the past decade lots of publications proposed deduplication of background storage as a viable form of data compression, especially for secondary storage systems to forward the replacement of magnetic tape systems. Most deduplication systems require significant amounts of main memory to provide fast accessible deduplication related metadata, making these systems very expensive. With the rise of fast and economically affordable Solid State Drives (SSD) some [3, 5] proposed to store deduplication metadata partially or entirely on SSD storage, reducing the in-memory structures.

The aim of this thesis is to evaluate, whether one could totally abandon any dedicated caches for deduplication metadata and still keep in acceptable performance. Cache-abandoning deduplication systems may be of particular interest for smaller organisations' data backup and archive purposes, as they suffer the most from high hardware costs due to enormous main memory requirements.

To evaluate the feasibility of cache-abandoning inline deduplication systems we designed and implemented an extension to the open-source filesystem ext4fs [9] providing inline deduplication, which we call *ext4fs+dedup*. This system was examined with several benchmarks and the results were compared to the unmodified ext4fs to determine deduplication overhead. Cache abandoning would for instance be infeasible, if it extends data transfer such that the system cannot meet the deadline of completion. The contribution of this thesis is determining the overhead of deduplication without the usage of any dedicated main memory cache. Our benchmarks showed deduplication overheads of 7 to 55 percent, dedpending on the workload.

This thesis proceeds as follows: In Chapter 2 we describe the design of our system. Chapter 3 discuses of implementation specifics. In Chapter 4 we present our evaluation setup and results. Chapter 5 provides an overview of related work and proposes a categorisation of inline deduplication systems. Finally, Chapter 6 concludes the thesis.

Chapter 2

Design

In this chapter we state our design goals and describe the alternative solutions. We explain our prefered concept and how we integrate it into ext4fs.

2.1 Design goals and code base

There are various systems, which provide deduplication of data. These systems differ mainly in their main memory and secondary storage consumption and the deduplication point in time.

In contrast to existing deduplication solutions with inline detection, our system does not use any dedicated main memory caches beside the buffer cache. Required deduplication information is read on demand from the target filesystem itself.

A filesystem provides complete deduplication if no block is written out when a block of the same content already exists on disk. Ext4fs+dedup provides such a complete duplicate detection. Every block written out to the filesystem will find a potentially present duplicate on disk no matter when or where the duplicate was written in the past. We call these detections duplicate hits. Figure 2.1 outlines the basic concept.

Furthermore, the design of ext4fs+dedup aims at an economical usage of storage resources. We wanted to keep the additional meta data as small as possible.

We based our design on the open-source filesystem ext4fs because of its actuality and its great deployment as well as because of its public sources.

2.2 Duplicate recognition

An efficient complete inline detection of duplicates depends on a fast recognition of already stored block content.

There are three basic recognition approaches:

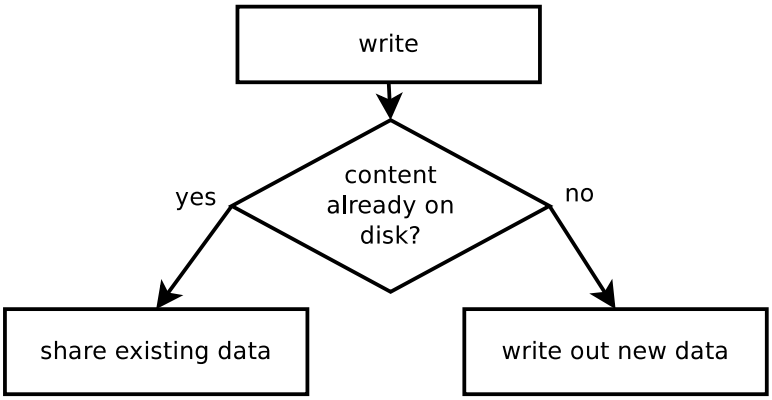


Figure 2.1: Outline of the inline deduplication concept.

- Naive: Linearly scan the whole filesystem and bytewise compare the to-be-written block to every used filesystem block.
- Ordered: The filesystem blocks are somehow (e.g., lexicographically) ordered according to their content on disk and thus one can use a binary search.
- Referenced: One remember where the filesystem stored one what content and maintain a content-location mapping.

The first two approaches are improper for our purpose: the naive one would cause enormous I/O traffic and thus fails to meet out efficiency needs. The ordered approach would have to frequently move written blocks to other locations to maintain the content order and therefore would stress the I/O device, too.

Consequently, we decided to realise the third approach. Therefore we need an information base that provides the content-location mapping.

2.3 Information bases for content-location mapping

In the beginning of our design phase, we discussed two approaches:

Content ordered binary tree For each used filesystem block there is a tree node located in the inode the block belongs to. The tree nodes are sorted (e.g., lexicographically by content). Figure 2.2 outlines the design.

However, the tree approach has some drawbacks:

- Tree search causes scattered I/O access, because the tree nodes are distributed over the filesystem.

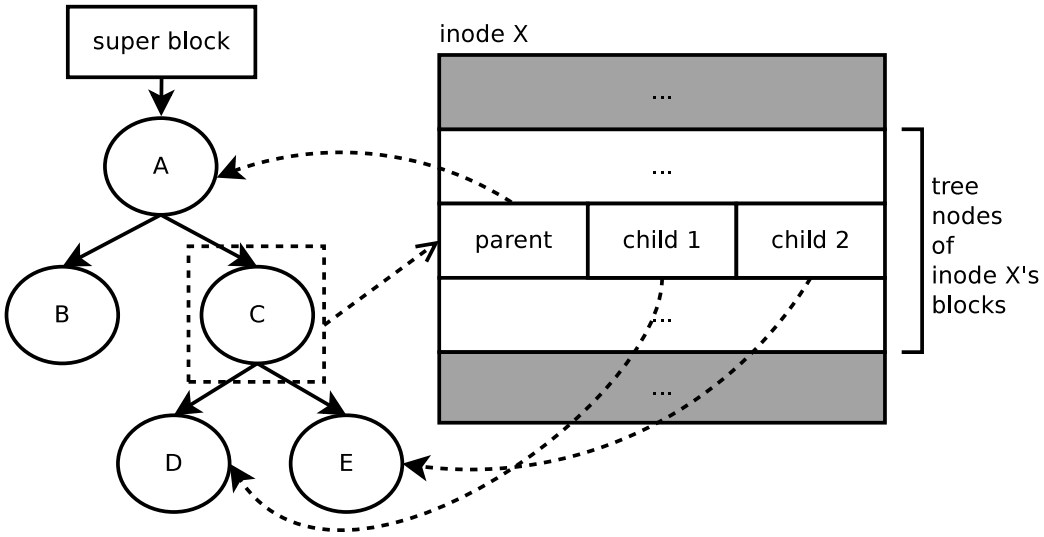


Figure 2.2: Outline of the ordered binary tree approach to map content to location.

- Deduplication causes shared blocks and therefore an unambiguous block to inode assignment is not possible.

- With the introduction of the extent feature, ext4fs trends to avoid inode meta data in block granularity. Putting the tree nodes into the inodes would thwart this trend.

Hash table This is a centralised array of content-descriptor to block number mapping with a static size. Figure 2.3 illustrates the basic hash table layout.

The drawbacks are the general hash table problems (e.g., collision handling, degeneration).

We decided to use the hash table approach, because of the crucial drawbacks of the tree design. Especially, the hash table allows—due to the grouping of mapping information—to benefit from spacial locality effects.

2.4 Fingerprint as content descriptor

To identify the correct mapping in the hash table, one can either match each candidate's content byte by byte with the to-be-looked-up content or one can use a handier content identifier. Block fingerprints generated with a cryptographic hash function like MD5 or SHA-1 are proper identifiers. They are highly injective and collisions are unlikely.

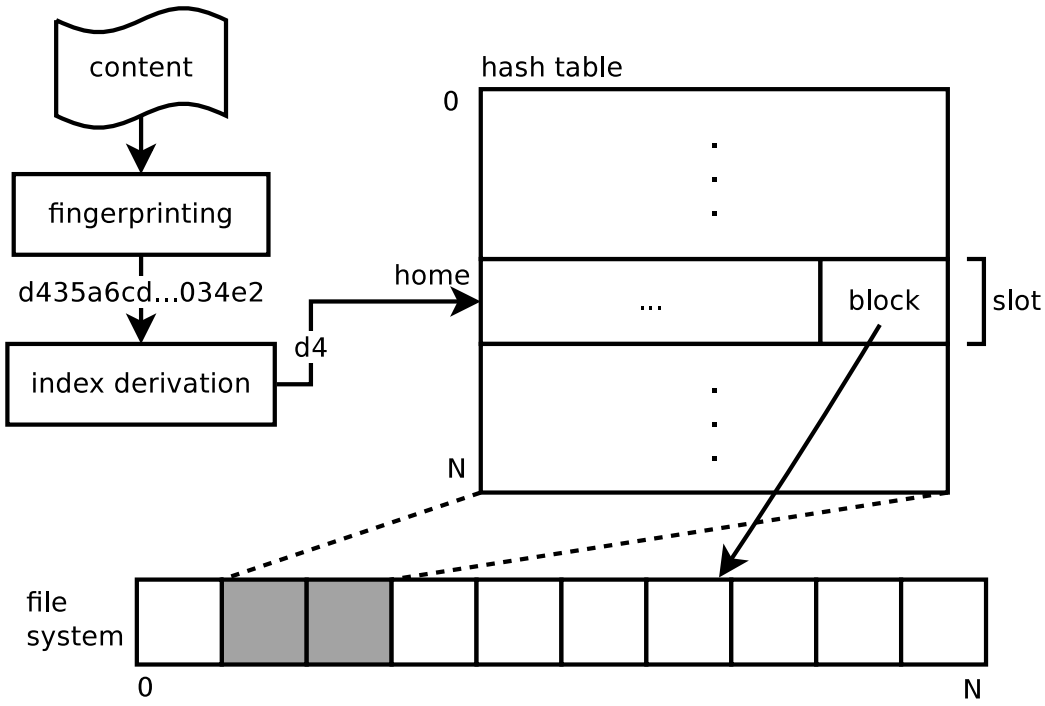


Figure 2.3: This figure shows the basic hash table layout and the home index derivation.

The fingerprints need to be stored permanently. Otherwise, we would have to gradually recalculate them during mount time, which contradicts the deduplication design goal completeness, because one can not detect a duplicate in a filesystem block, that was not yet accessed during the current mount period.

We chose the cryptographic hash algorithm SHA-1 to fingerprint the block contents because of its common usage and support within the linux kernel. It produces 160 bit long hashes. This length seems to be sufficient, because a hash collision of two different contents is very unlikely (about 2^{-80}, c.f. [12]).

2.5 Hash table

The hash table is subdivided into a number of slots, corresponding to the number of filesystem blocks. Each slot has an identifier—we call it index. Figure 2.3 illustrates the subdivision into slots.

To locate the hash table slot, that contains the requested mapping to the block number of the potential duplicate, a slot has a special meaning.

2.5.1 Home slot

The home slot is the starting point of the lookup and the first possible mapping location. The derivation of the home slot's index is simple: the first 64 bits of the fingerprint are extracted and that value modulo the filesystem size in blocks is used as index.

$$home = fingerprint_{0-63} \mod \sharp fs\_blocks$$

This derivation is not injective and there are much more different SHA-1 fingerprints (2^{160}) than hash table slots. Therefore the home slot derivation of two fingerprints can collide.

2.5.2 Collision handling

All fingerprints, whose 64 bit prefix modulo the amount of filesystem blocks are mapped to the same home slot. This set of fingerprints is called a family. Due to the fact, that one slot can only be occupied by one fingerprint, every later written block with a fingerprint of the same family causes a collision and needs to be located somewhere else. To keep the records close to their home slot—thus take advantage of spacial locality effects—we use the next free slot after the home (linear probing). At this location the fingerprint occupies the home slot of another family of hashes. Therefore we call such occupants "strangers" (see Figure 2.4).

Without any further information it would be hard to find strangers during a hash table lookup. If the fingerprint is not present in its home slot, one would have to linearly scan the hash table to find the stranger. But one can never know if it is in the next slot or not until you scanned the whole hash table.

Instead, we link all strangers into a linear list anchored in the home slot. For that purpose every hash table slot has a 64 bit field to hold the hash table index of the next member in the family list, the `next` reference. Additionally, every slot has another reference field required for those slots, which are occupied by strangers to point to the family of fingerprints actually having their home in that slot. This is called the `family` reference. Figure 2.5 shows a hash table lookup, which involves the `family` reference, because the corresponding home slot is occupied by a stranger.

To indicate whether the next or the family reference fields contain valid indices as well as to label the occupancy status of a slot, we use a 8 bit status field per slot.

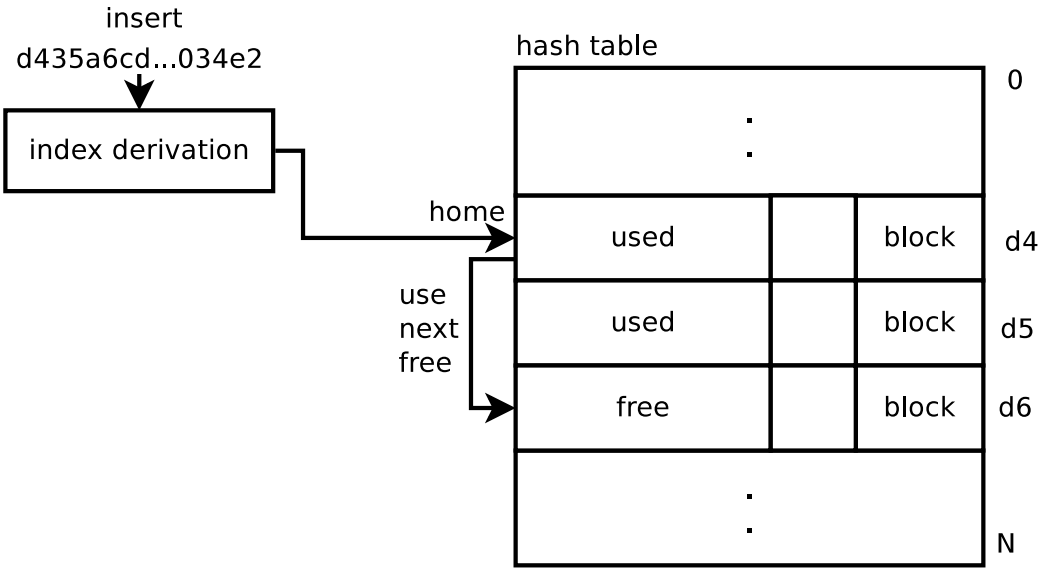


Figure 2.4: Inserted record uses a foreign home slot as stranger, because its own home slot is already occupied.

2.6 Deduplication Information Area

There are filesystem operations that require updates of deduplication related metadata but have no knowledge about the content of the affected filesystem blocks. Thus they cannot calculate the fingerprints unless they read the context from disk, which would cause extra I/O overhead plus the CPU overhead for the actual hash calculation. But the fingerprints are essential to determine the index of the related hash table records.

Such an unaware operation for example is the block deletion. If a filesystem block is deleted, the related hash table record has to be removed, too. Otherwise it would potentially cause false duplicate hits on such orphaned hash table records. Without additional information every deletion would cause an extra I/O operation for the fingerprint calculation. The same is true, if we would modify an existing block. Therefore the fingerprint would change and a removal of the old hash would be necessary, too.

Instead of an expensive read operation and hash recalculation, we put extra mapping information into the filesystem, to allow a linkage (backlink) of each used filesystem block to its corresponding hash table record.

We decided to use a dedicated centralised data structure similar to the hash table, which we call Deduplication Information Area (DIA). The DIA contains one record per filesystem block. But unlike the hash table, it is directly indexed by the physical block numbers. However, we considered alternative approaches,

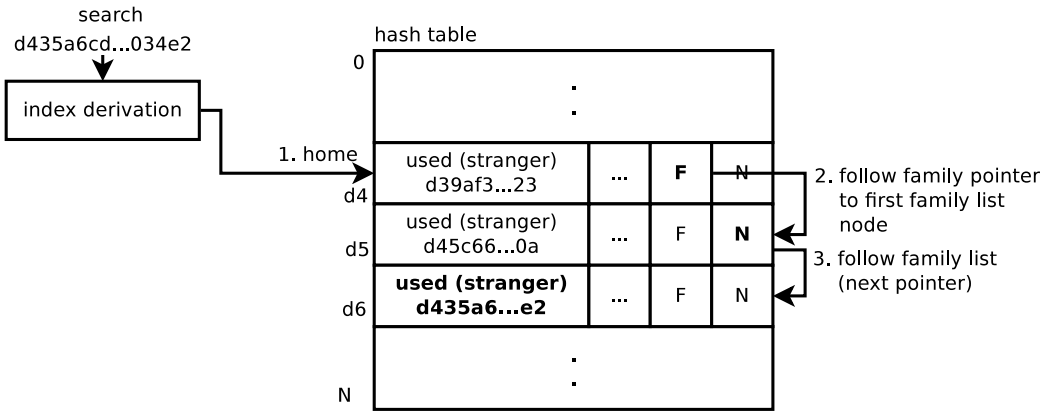


Figure 2.5: This hash table lookup illustrates the usage of the `family` and `next` references. The records within the slots `d5` and `d6` form a family. Their home slot is occupied by a stranger.

which we depict in the following section.

2.6.1 Backlink alternatives

The following three solutions provide different ways to retrieve the hash table record associated with a given filesystem block without knowing the block's content.

Solution 1: Backlinks in inode

This solution does not require an additional data structure like the DIA, but it puts backlink references into the inodes. For each filesystem block, that is mapped to the inode we store the index of the related hash table record within the inode. Figure 2.6 outlines the backlink in inode design.

This approach only make sense if the inode already contains per block informations. In this case you could extend the existing records by a 64 bit reference. However, due to the introduction of extents—in contrast to the old indirect block based mapping mechanism—ext4fs no longer maintains per block information within the inodes.

Therefore we buried that approach, especially as it does not work with one of the ext4fs core features.

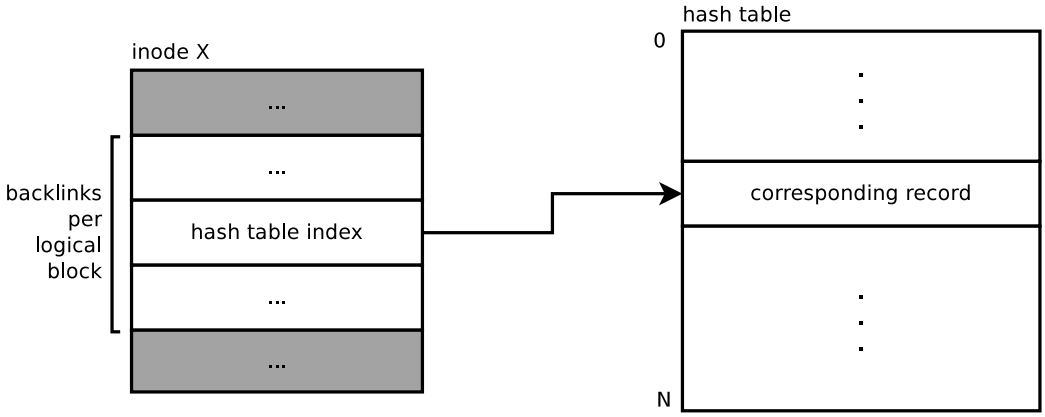


Figure 2.6: Backlinks in the inodes could allow the retrieval of hash table records for content unaware operations.

Solution 2: DIA with fingerprints (Large-DIA)

The second solution stores the fingerprints in the additional Deduplication Information Area. With that information, such fingerprint unaware operations can indirectly retrieve the related hash table record. Thereto, one have to derive the home index from the fingerprint and then search the corresponding record in the family list (see Figure 2.7).

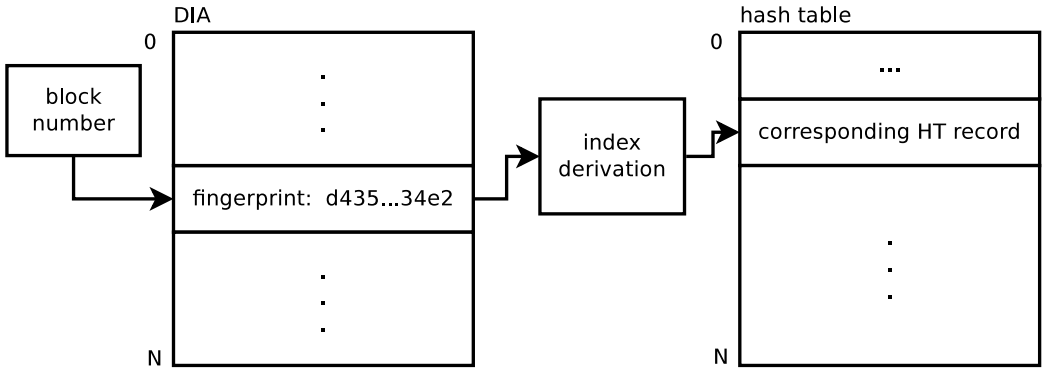


Figure 2.7: Indirect backlink via fingerprints in the DIA.

Solution 3: DIA with backlink (Slim-DIA)

The third solution uses the DIA as well, but as a container for the direct hash table backlinks described in Solution 1. Thereby we save the index derivation and especially the search within the family list (see Figure 2.8).

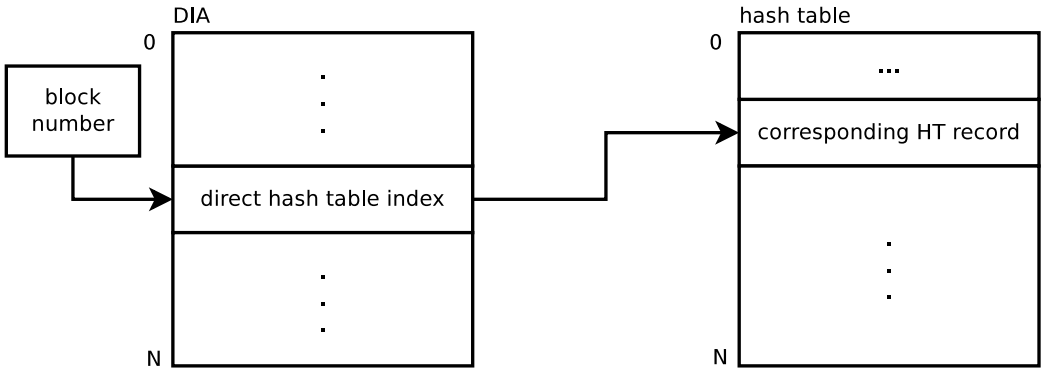


Figure 2.8: Slim-DIA is a central array of direct hash table backlinks.

For later reference we call the second solution Large-DIA whereas the third solution is called Slim-DIA.

Comparison

Regarding the memory consumption, Slim-DIA requires more storage than Large-DIA, namely that part that stores the backlink references, which are not present in the other solution at all. Slim-DIA does not economise the storage for the fingerprints. The fingerprints are required in any case to prove the matching of the found hash table record. Since the hash table index covers only a part of the fingerprint, it is essential to store the complete hash somewhere to prove a full match. If and only if the complete stored fingerprint matches that one of the looked up block, we can assume a duplicate hit.

But in terms of required I/O operations the Slim-DIA is advantageous for hash table lookups. As we saw above, a complete fingerprint must be present in memory to do a full match against the fingerprint given in the lookup request. That match is required for every record that is linked into the family list.

Since the Slim-DIA stores the fingerprints directly in the hash table record itself, they are read together with the references, block number and complete record. No additional I/O request is needed. The Large-DIA, that stores the fingerprints in the DIA, requires an additional fetching of the block containing the DIA record with the fingerprint. Without any further hash informations (cf. big tag vs. status tag 2.8) these fetches are required for every family list member—at least until we found a hit.

Beforehand, One more—not deduplication related—advantage of Large-DIA can be adduced. If later implementations would optionally use the fingerprints for data integrity checking, then the direct retrieval out of the DIA would be easier and faster than the indirect backlink approach. The read of the related DIA block

could be issued together with the read of the actual data block. But that's not the subject of this thesis.

2.7 Reference counters

As a result of deduplication, a physical filesystem block can be referenced multiple times. Thus that block has to remain untouched unless the last reference is unmapped (e.g., inode truncate). To determine last unmap, we use per block reference counters comparable to those used within inodes to support hard links.

2.7.1 Possible locations of the reference counters

We have two different data structures involved during the deduplication process, and considered both for the location of the reference counters. The respective strengths and weaknesses of each location depend on the used DIA design (Large-DIA or Slim-DIA).

Ext4fs+dedup uses Large-DIA and thus locates the counters in the DIA. This location saves a filesystem access compared to the hash table location. To understand that fact, let's have a look on the circumstances of reference counter updates.

When and why are reference counters accessed?

A reference counter is incremented whenever a duplicate block is found and initially set to one when the block is first written. In both cases we already know the block fingerprint, as we previously calculated it for duplicate detection.

Furthermore, the counter is decremented whenever a block is deleted. When deleting a block, we neither know its content nor its fingerprint.

Regarding the HT location, the latter access case would require the indirection over the DIA to derive the hash table index by backlink or by fingerprint.

Favoured locations

A favourable location saves I/O operations. Table 2.1 shows the required reads and writes for each operation that access the reference counters and each DIA design variation and the reasons why they were issued.

Accesses after writing of new blocks

Writing an unique block requires an update of all related records. A new hash table slot is occupied and its index respectively the fingerprint is put in the DIA record

design / operations	write unique block →HT insert	duplicate hit →refcnt inc	block deletion →refcnt dec	total
Large-DIA refcnt in HT	HT: `r(s)+w(snc)` DIA: `w(f)`	HT: `r(nc)+w(c)` DIA: `r(f')`	HT: `r(c)+w(c)` DIA: `r(f')`	9
Large-DIA refcnt in DIA	HT: `r(s)+w(sn)` DIA: `w(fc)`	HT: `r(n)` DIA: `r(fc)+w(c)`	HT: – DIA: `r(c)+w(c)`	8
Slim-DIA refcnt in HT	HT: `r(s)+w(snfc)` DIA: `w(b)`	HT: `r(fc)+w(c)` DIA: –	HT: `r(c)+w(c)` DIA: `r(b)`	8
Slim-DIA refcnt in DIA	HT: `r(s)+w(snf)` DIA: `w(bc)`	HT: `r(fn)` DIA: `r(c)+w(c)`	HT: – DIA: `r(c)+w(c)`	8

Access reason abbr.	meaning
b	backlink reference (Slim-DIA)
c	reference counter
f	fingerprint for full match
f'	fingerprint only for HT entry retrieval (Large-DIA)
n	block number to retrieve DIA record
s	hash table record status (esp. occupation status)

Table 2.1: The upper table lists the required HT and DIA accesses for each operation and design. Within the parentheses, a combination of one letter abbreviations indicates the reason for the read (r) or write (w) access. These abbreviations are described in the lower listing.

of the used physical filesystem block. This case does not favour any location no matter which DIA design we use.

Accesses after a duplicate hit

In the case of a duplicate hit, the counter is read and incremented. This at least requires an update of the record containing the reference counter. Furthermore reading the hash table record has already taken place during the duplicate lookup. To verify matching fingerprints, the stored hash is fetched from its location. At this, Slim-DIA favours the hash table as location for the reference counters, because if so, all required information (esp. fingerprints and counters) are bundled in the hash table records. Accessing the DIA is then not required at all. However, Large-DIA already demands fetching related DIA records for matching fingerprints. Thus the counter location makes no difference in respect of the amount of required filesystem reads and writes.

Accesses during block deletion

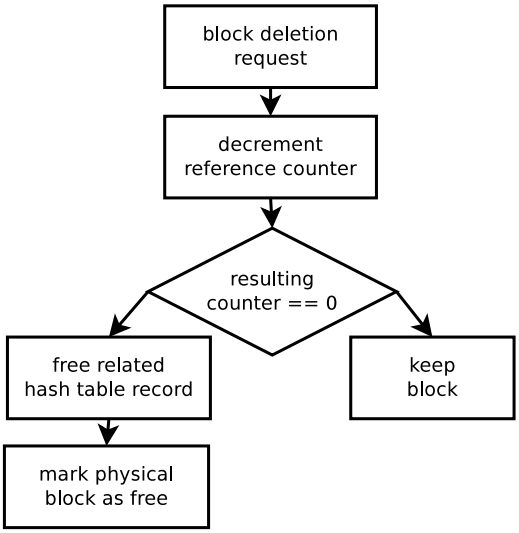


Figure 2.9: This flowchart illustrates deduplication related housekeeping after a block deletion request.

The remaining case of a block deletion can be subdivided according to the resulting reference count after the decrementation (see Figure 2.9). If the last reference is removed and the counter is zero, a status update of the related hash table record is mandatory to avoid later duplicate hits on the orphaned record. If the decremented counter is greater than zero, the block is still in use and the hash table record can remain untouched, provided that the reference counter is not located in the hash table. Putting the counters into the hash table would require a read and write of the hash table record, but also a read of the DIA record to derive the hash table index. Therefore the Large-DIA as well as the Slim-DIA favours the DIA as the location for the reference counters, because it saves the second read operation in the case of a non-final block reference deletion.

2.7.2 Conclusion

Putting it all together, the Large-DIA approach clearly favours the DIA as the location for the reference counters, because we save on filesystem access, whereas the hash table location doesn't permit any savings. Regarding the Slim-DIA approach, the best reference counter location is not obvious. Both locations save one access. We consider the hash table to be the better choice, because the saving effects the block write path and there an increased performance may be beneficial to the user. Furthermore the effects of the savings depend on the filesystem

use cases. For example a backup system with an eternal history would frequently cause duplicate hits but never cause block deletions.

2.8 Additional fingerprint information in the HT

The major drawback of the Large-DIA approach are frequent DIA access during hash table lookups, that are required for fetching the fingerprints needed for full fingerprint comparison. The more hash table collisions occur, the longer family lists will be and hence the more such comparisons are required.

To mitigate this drawback, additional parts—called tags—of the fingerprints can be placed in the hash table. Then the complete fingerprint only needs to be fetched if the tag matches the corresponding part of the looked up fingerprint, otherwise the complete fingerprints wouldn't match either anyway.

The amount of actual full fingerprint comparisons can be influenced by the length of the fingerprint tag in the hash table. The longer the tag, the less frequent the comparisons are.

2.8.1 Implemented tag variations

We implemented and evaluated two variants of fingerprint tags. The "status tag" uses the remaining bits of the status field to store a few fingerprint bits. Whereas the "big tag" uses an extra 8 byte field. These 8 bytes fill up the hash table record to the next power of two, which is beneficial in terms of block alignment.

However, big tag thwarts the storage savings of the Large-DIA approach compared to the Slim-DIA. Nonetheless we implemented big tag, to show the degree of performance influence of the tag length.

Figure 2.10 shows the resulting hash table and DIA layouts of the chosen Large-DIA approach.

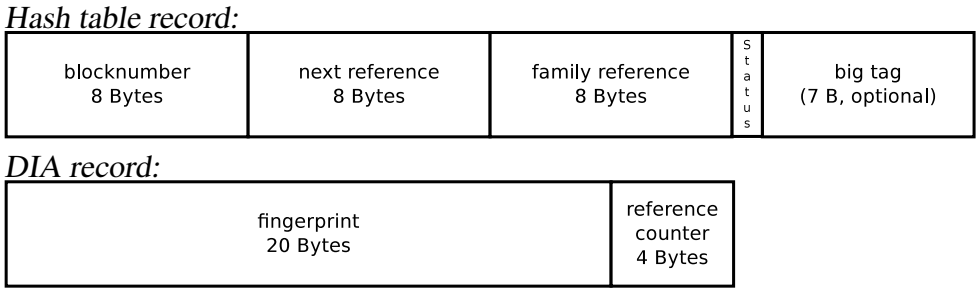


Figure 2.10: Hash table and DIA record layouts for Large-DIA and both tag variants.

2.9 Storage overhead

Depending on the used tag variant, ext4fs+dedup causes at least 1.2 percent of meta data overhead in order to deduplicate data blocks. Table 2.2 shows the total amounts of storage used for the hash table and the Deduplication Information Area. The listed percentages assume the common block size of 4 KB and are calculated as follows:

$$dedup\ data\ percentage = \frac{HT\ record\ size + DIA\ record\ size}{block\ size}$$

	status tag	big tag
HT record [bytes]	25	32
DIA record [bytes]	24	24
Total per block [bytes]	49	56
Percentage	1.20	1.37
Deduplication data for a 1 TB filesystem [gigabytes]	12.25	14

Table 2.2: Total storage overhead caused by meta data for deduplication.

2.10 Duplicate management operations

This section explains the various operations that are related to the detection and management of duplicates. For each operation we describe what they do and the circumstances under which they are issued. Note that the follow explanations are based on the Large-DIA approach.

2.10.1 Duplicate lookup

First, ext4fs+dedup calculates the fingerprint for every block that is written to the filesystem. Out of the fingerprint, it derives the hash table index of the home slot and then fetches the corresponding hash table block. If the status field indicates a valid family reference then we follow the reference to retrieve the head of the family list to start the list walk. Thereby the home slot can be either occupied by a stranger or can be free, anyway its not part of the family.

Otherwise, if the status field indicates an occupation by a family member we can immediately walk the family list.

But if the status field labels the slot as free and the family reference is not set, then we can stop the lookup and report that the requested fingerprint is not present (cf. Figure 2.11).

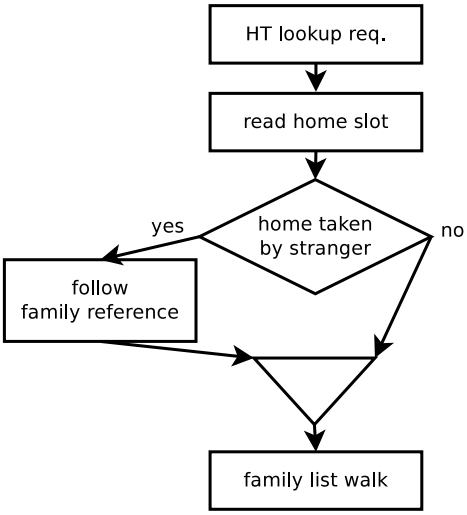


Figure 2.11: A stranger may occupy the home slot. Then the family reference points to the first family member.

Family list walk

After ext4fs+dedup got the record of the family list head, it compares the fingerprint tag (status tag or big tag) to the given fingerprint. If they match, it does a full fingerprint comparison fetching the fingerprint from the related DIA record, otherwise it goes on to the next family list member referenced by the `next` field. Ext4fs+dedup repeats this procedure until either the full fingerprint comparison succeeds or it reached the end of the family list reference.

If ext4fs+dedup found a matching fingerprint during the family list walk, then it reports the physical block number, that is stored in the found hash table record. Figure 2.12 depicts the family list walk.

2.10.2 Insertion of a new and unique block

A block is unique if the previous duplicate lookup found no duplicate in the filesystem. In this case ext4fs+dedup remembers the calculated fingerprint. Later during write procedure (cf. 2.11) it puts that fingerprint and the newly allocated block in the hash table and the DIA. Therefore the index of the home slot is derived and if this slot is already occupied we use the next free slot after the home slot and make it the new head of the family list. The counter is initialised to one.

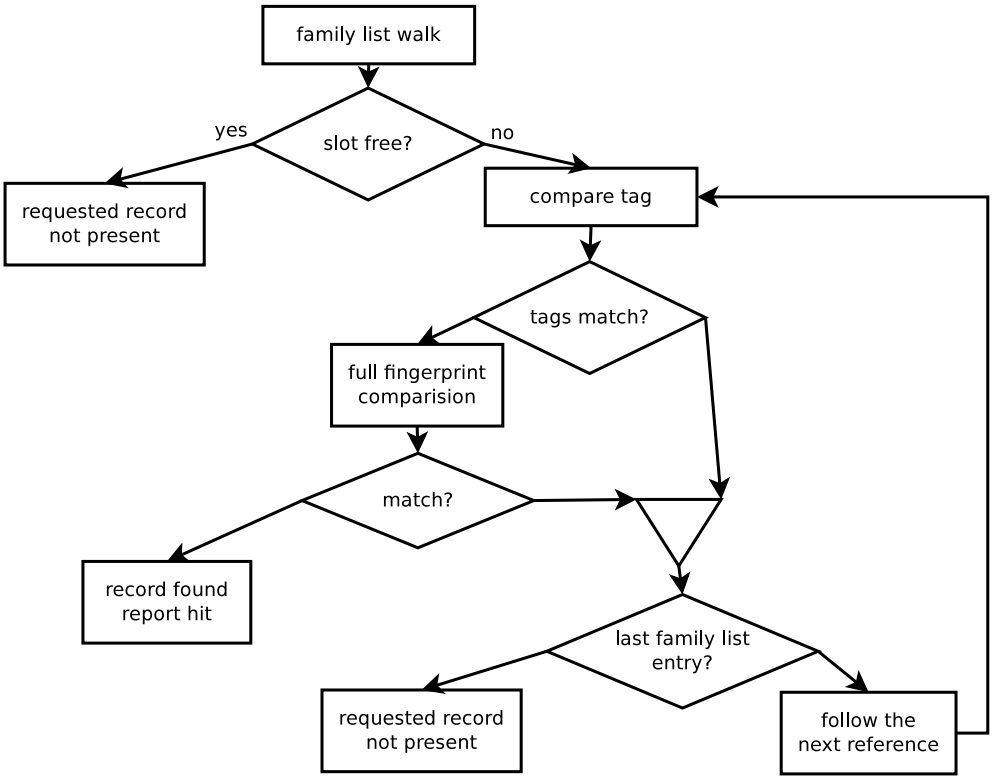


Figure 2.12: A family list walk follows the `next` reference until either the full fingerprint comparison succeeds or no further list member is indicated.

2.10.3 Insertion of a duplicate

Every time a duplicate is found and used the reference counter needs to be updated. Therefore the related DIA record is fetched to read the old reference counter value. After the incrementation, the counter is written back.

2.10.4 Deleting blocks

Block deletions require at least the decrementation of the reference counter, which is executed similar to the counter incrementation described above. Only after the deletion of the last reference, further housekeeping needs to be done.

Removal of last reference

The last reference is removed if the counter is decremented to zero. Then the corresponding record within the hash table has to be removed, too. We look this record up by using the duplicate lookup routine described above. It is not required

to compare the full fingerprints, because in consistent and uncorrupted hash tables there is only one non-free record with the required block number reference. However, for the sake of simplicity and also for consistency checking we compare fingerprints anyway.

Note, that the comparison of the block number registered in the hash table record to the block number requested for deletion is essential. As it is possible that there are multiple valid hash table records for the same block content but for different physical blocks. This happens, if these blocks are flushed to the filesystem at the same time and thus no duplicate can be found since none of them is actually written yet and ext4fs+dedup does not detect duplicates within the buffer cache (cf. Figure 2.13). Therefore, it is insufficient to simply remove the first record, that has a matching fingerprint, but to regard the block number field too.

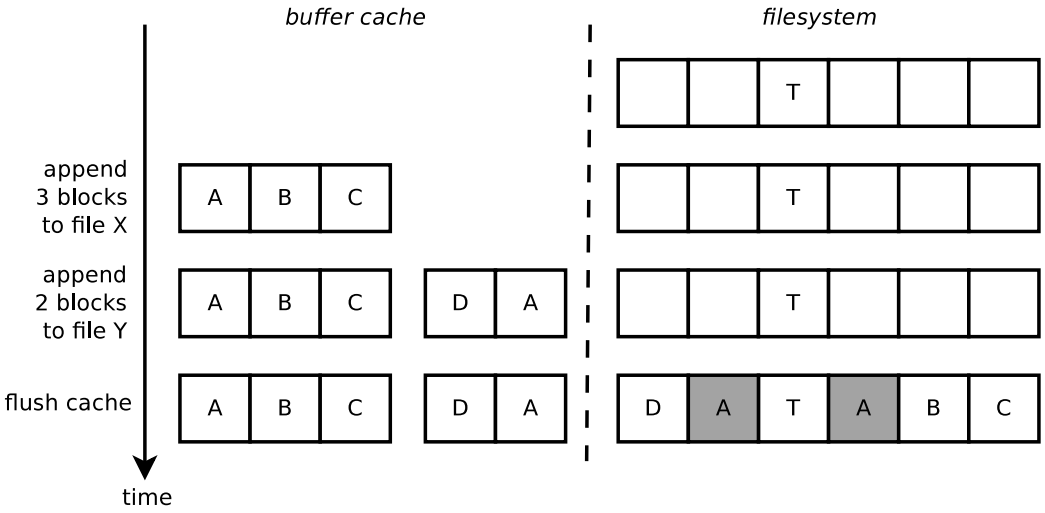


Figure 2.13: If identical blocks are flushed at the same time, the duplicate recognition fails to detect them.

2.11 Integration into ext4fs

Ext4fs has some features that are advantageous for inline deduplication. Especially the delayed allocation harmonises well with our deduplication process. Without delayed allocation ext4fs and its predecessors allocate new filesystem blocks at the begin of a write command, in others words the blocks are allocated before the content is copied to the buffer cache and before the filesystem can see the content. Later on, a detected duplicate within this content would save one

filesystem block. Since the blocks are already allocated, the allocation has to be undone again.

Delayed allocation makes this rescission unnecessary. The allocation is delayed to the end of the write. At this point the to-be-written data is already present in the buffer cache and thus accessible to the filesystem procedures. With this additional knowledge, duplicates can be detected before allocation took place.

In the following, we describe the integration of the deduplication ability into the delayed allocation code path and the adaption of the extent management and the block allocation by portraying the flow of a block through the code path.

2.11.1 Block write

The block writing path is complex. Figure 2.14 illustrates the following descriptions of the basic integration concept.

Grouping to extents

An extent is an area of physically adjacent filesystem blocks that is originally dedicated to one inode to store logically adjacent file blocks. Thereby the amount of mapping meta data per inode is reduced to a few bytes per extent (logical offset, physical offset, length) instead of one physical block number per logical block.

Therefore, at the beginning of the delayed allocation's write end handling, the to-be-written blocks within the buffer cache are grouped to extents. Blocks, that are locally adjacent and share the same properties are bundled to an extent.

We extended this functionality by a more detailed property selection. Therefore each block (represented by a buffer head) is passed to the duplicate detection. The found duplicate blocks are labeled accordingly and the detected block number is remembered within the buffer head structure.

The non-duplicate blocks have to be actually written to the filesystem. These blocks can either be unmapped or already mapped. But a mapped block may have previously been subject to deduplication and thus may be shared. Such a shared block must not be directly overwritten, because the modification should only effect the issuing inode and not the other ones that reference that block too. Therefore we also label such shared blocks.

Now, these two deduplication specific block properties are also taken into consideration. As a result, only adjacent blocks of equal properties—inclusive the duplicate status—are grouped together. Especially, two detected duplicates can only be bundled, if they are not only logically, but also physically adjacent, because the to-be-shared physical filesystem blocks are already mapped, but may be located within totally different filesystem regions. In the latter case such duplicates cannot

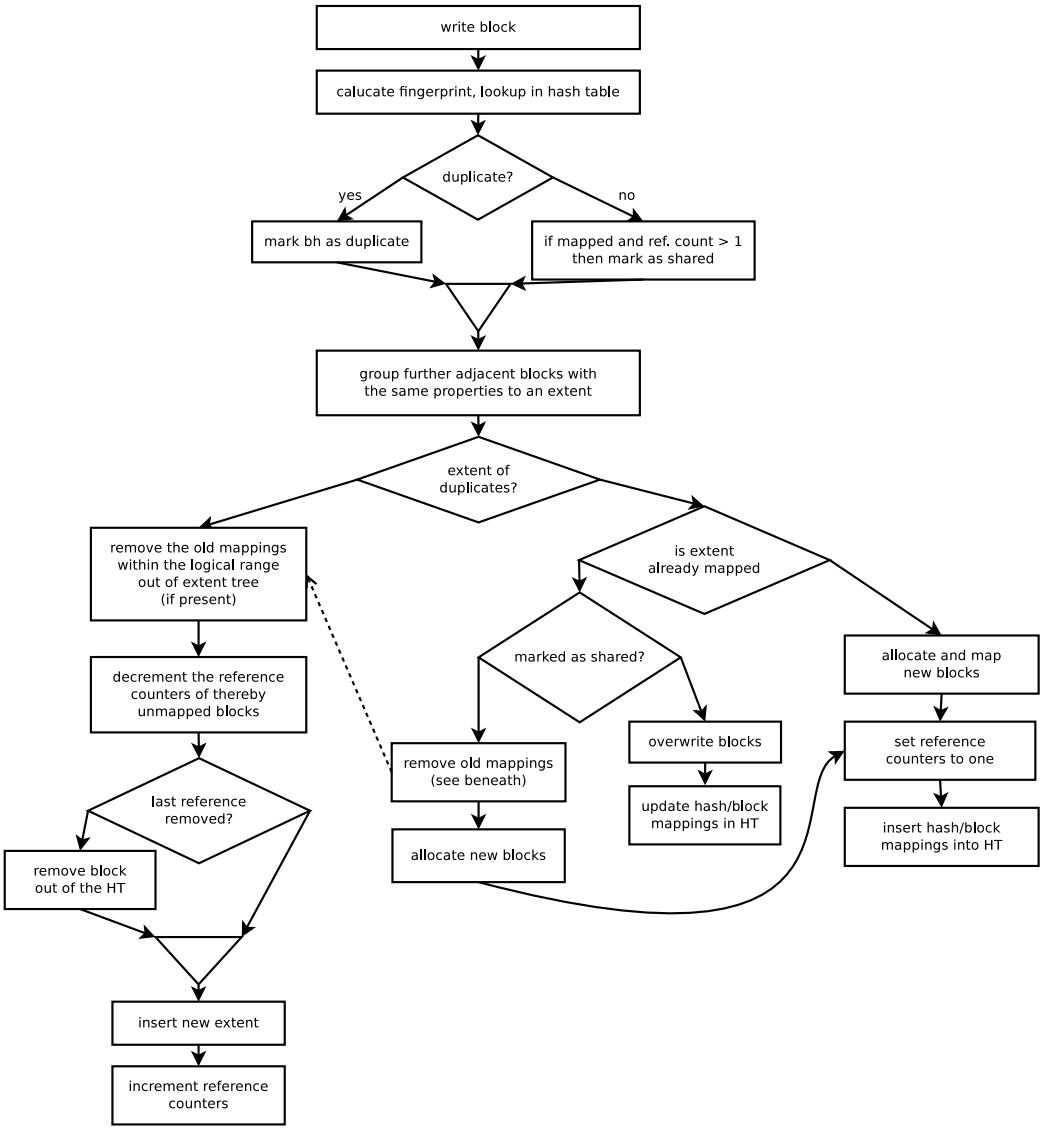


Figure 2.14: Basic deduplication procedure during a write.

be merged to one extent, but have to be registered as two separate extents in the inode's extent tree.

Those dirty blocks, that are already mapped and unshared can be immediately submitted for writing. No further allocation or mapping is required and an in-place overwrite is possible.

Extent mapping

The collected extents are inserted into the inode's extent tree. But previously, those extents that weren't already backed by physical filesystem blocks as well as those that were shared have to allocate new blocks. The shared ones need them for the so called 'copy on write' (COW).

At this point we distinguish three further methods.

- Duplicates: If the extent is a group of duplicates, then an allocation is unnecessary. We just remove old mappings from the extent tree and insert the new extent.
- Shared: In the case of a shared extent, the old mapping is removed and the extent pointing to the newly allocated physical blocks is inserted into the extent tree. Now the blocks are submitted to the block layer for the actual writing.
- Else: In all other cases the allocated blocks are inserted and are submitted for writing.

2.11.2 Block unmapping and removal

Block removal is requested if an inode is truncated. Then the affected blocks are removed from the extent tree and marked as free.

Due to deduplication we have another reason for removals: the detection of a duplicate for a logical offset, that is already mapped and the consequent unmapping of the old referenced block.

However, because of block sharing, no unmapping is allowed to cause a block free until the last reference is deleted. Therefore ext4fs+dedup intercepts every free request, to decrease the reference counter of every requested block. Only those blocks, that got their last reference removed are actually passed to the freeing routine.

Chapter 3

Implementation

This chapter describes implementation details, especially difficulties that we have faced during the implementation. Furthermore, we think about the effects of the hash table occupancy rate and discus the drawbacks of the extent feature in combination with deduplication.

3.1 Preallocations and doubly freed blocks

During first tests we have faced several consistency problems related to the free blocks management. Discarding ext4fs preallocated spaces when handling duplicates caused, that blocks were freed twice.

Ext4fs uses a sophisticated block allocation mechanism to avoid fragmentation. Preallocation is one fragmentation avoiding mechanism, which is based on the insight, that data is often appended to previously written files. Therefore, the allocator speculatively allocates more contiguous storage than requested enabling later appends to be put in this preallocated area instead of into distant storage regions. These preallocated spaces have an ambiguous allocation state. On the one hand they are free, for they don't contain valid data yet. On the other hand they are occupied, for they are dedicated to a single inode and not available to subsequent allocation requests. Consequently, ext4fs introduced an in-memory data structure to handle this special allocation states.

Preallocation lists

Preallocated space lists are maintained per inode as well as per block group. Therein, preallocated storage chunks are registered, which are leftovers, that accrue when the allocator allocates more space than requested. This overallocation is called normalisation and should optimise the allocation request in terms of size

and alignment. It is a speculative preallocation, which intends to speed up later requests and moderate fragmentation. Figure 3.1 illustrates an normalised allocation request and the resulting preallocated space (PA).

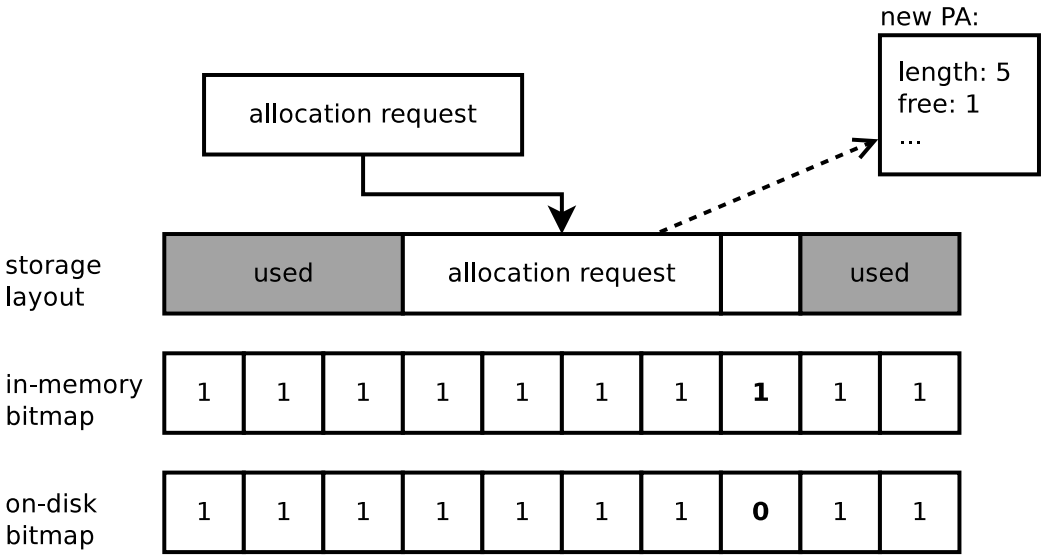


Figure 3.1: Due to normalisation, the allocation request is over-served by one additional block resulting into a new preallocated space (PA) which is registered in the per-inode PA list. Regarding this preallocated block, the two block bitmaps differ, for preallocations are not persistent.

Note, that preallocation lists have nothing to do with the user preallocation done via the *fallocate* interface. The multiblocks allocator's preallocations are filesystem internal and not visible to the user at all.

Non-persistent allocation information

Ext4fs as well as its predecessors use persistent on-disk bitmaps to distinguish used from free blocks. In addition, ext4fs uses non-persistent, in-memory bitmaps that label additional blocks as occupied, namely those that are preallocated but not used yet. In Figure 3.1 the one preallocated but yet unused block is occupied according to the in-memory bitmap but free according to the on-disk bitmap. The following equation describes the quantitative relation of marked-used blocks between the on-disk bitmaps and the in-memory informations.

$$\textit{on-disk bitmap} = \textit{in-memory bitmap} - \textit{free blocks in preallocations}$$

If a request is normalised and served with more blocks than requested, then the complete block range—not only the excessive part—is inserted into the preal-

location list of the requesting inode—or in the locality group list (we don't discus the latter to keep it clearer).

The in-memory bitmaps mark the whole preallocated space as reserved and the per-inode preallocation list exclusively dedicates it to the requesting inode. No other inodes can allocate these blocks even if they are not in use yet, but marked as free in the on-disk bitmap.

Discarding of preallocated spaces

Preallocations are originally discarded in the following situations:

- The filesystem runs out of free blocks.
- An inode is truncated.
- The filesystem is umounted.

Discarding marks the unused blocks of the preallocation space as free in the corresponding in-memory bitmap. However a preallocated space list entry only stores the amount of unused blocks within the block region and not the position of these free blocks. To distinguish used from unused PA blocks, ext4fs reads the corresponding on-disk bitmap, carries the free marks into the in-memory bitmap, and deletes the PA list entry.

3.1.1 Problem

Originally, ext4fs does not partially free used blocks within an inode, but only frees used blocks if the corresponding inode is truncated. In the latter case ext4fs also discards all preallocations dedicated to that inode.

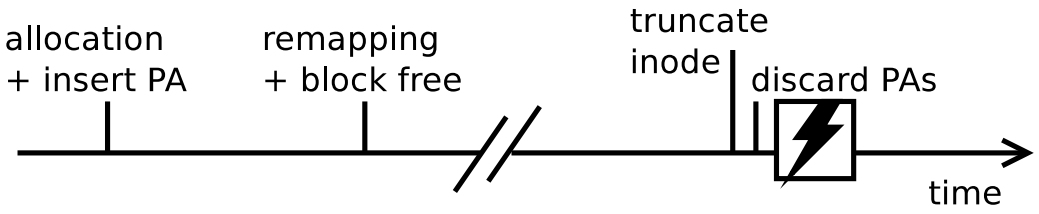


Figure 3.2: Timeline of a double free error.

Ext4fs+dedup additionally frees blocks when remapping them to shared blocks after duplicate detection, provided that the previously mapped block has no further references, hence is unshared. In this case ext4fs+dedup frees the block and marks it as free in the in-memory and on-disk bitmaps. If such a freed block was

part of a preallocated space, then a later discard of that PA would free the block again (see Figure 3.2).

The following chain of filesystem events leads to a double free:

1. Inode A requests the allocation of n blocks, starting at logical offset l. The request is normalised and $n+k$ blocks starting at the physical block number p are allocated. A preallocated space is registered (cf. Figure 3.3).

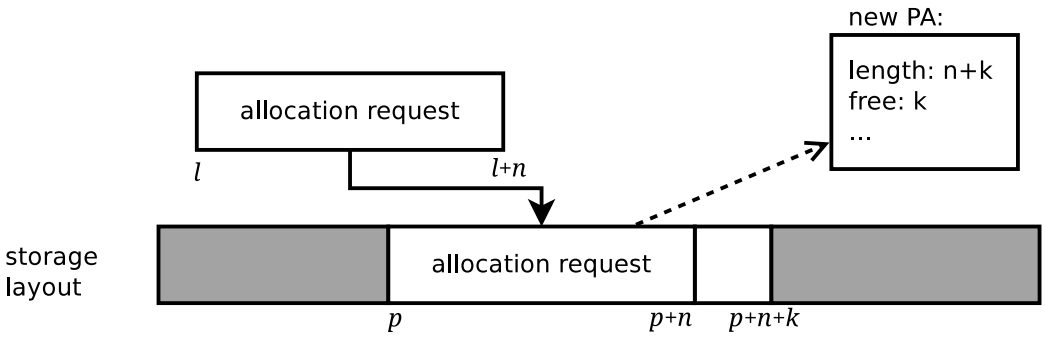


Figure 3.3: The allocation request has been served from physical storage offset p. Normalisation led to a preallocation of k blocks.

2. A rewrites one block at logical offset l and a duplicate is detected at d. Ext4fs frees and unmaps the unshared physical block p. Logical offset l is mapped to d (see Figure 3.4).

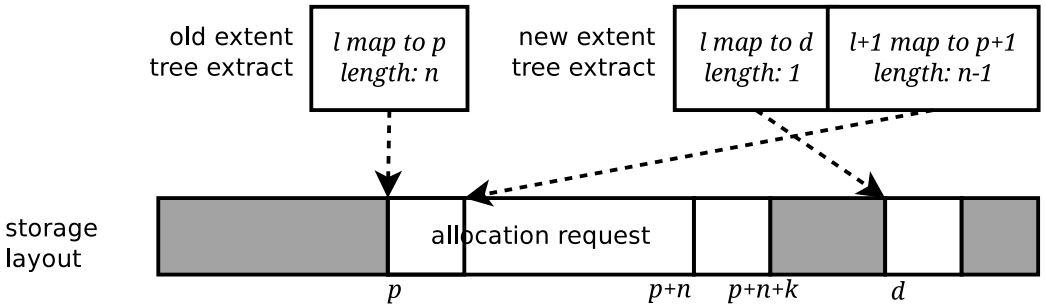


Figure 3.4: Due to a duplicate detection, physical block p is no longer required. Block d is mapped instead.

3. Later the inode A is truncated and its PAs are discarded. A still has the PA that covers the blocks p to $p+n+k$. Ext4fs tries to free PA's k preallocated but unused blocks in the in-memory bitmap. Therefore all blocks within in the PA range, that are free in the on-disk bitmap are freed in the in-memory bitmap.

4. Scanning the on-disk bitmap, ext4fs also tries to kill the bit of block p. But the in-memory bitmap bit of block p is already killed, because p was explicitly freed in-memory and on-disk before during the remap. Hence we face a double free. This is an unexpected inconsistency and ext4fs treats this as an error.

3.1.2 Solution

Double frees can be avoided by discarding related preallocated spaces during the remap procedure. It is important to discard the PA covering block p (the old and to-be-replaced block) before p is freed, thereby ext4fs+dedup delays the killing of p's on-disk bitmap bit to avoid a misinterpretation by the PA discard procedure, which would otherwise cause the second free. Figure 3.5 outlines the previous scenario.

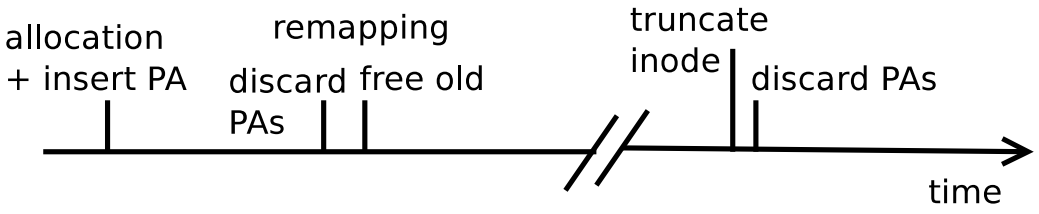


Figure 3.5: Timeline with avoided double free problem.

Discarding all preallocated spaces

Ext4fs+dedup does not only discard the PA of deduplicated blocks, but every preallocation that is dedicated to the corresponding inode. Discarding PAs is already implemented in ext4fs and used during inode truncation, so it was easy to reuse this functionality in the context of remapping.

3.2 Initialisation of hash table and DIA

An correct initialisation of the hash table (esp. the status field) and the DIA (esp. the reference counter) is essential for ext4fs+dedup. Blocks, that are dedicated to these data structures have to be zeroed out before usage.

Ideally, this happens when the filesystem is created, however the `e2fsprogs` do not support extents so far (Version 1.42.3). Since we wanted to use extents for the (hidden) inodes representing the hash table and the inodes, we looked for a workaround.

We tried two approaches: The first tried to zero out the blocks on demand and used the *fallocate* feature for preallocation. The second one allocates and zeroes out all blocks en bloc once the first access happens. In the following we describe both approaches. We have finally chosen the second work around due to lower metadata management overhead.

3.2.1 Zero out on demand

On the very first access to the hash table or DIA, the respective data structure is preallocated. The preallocation is necessary to guarantee sufficient storage for deduplication structures. We statically (pre-)allocate the maximally required number of blocks. Thereby we are prepared for the worst case and furthermore we avoid fragmentation.

The preallocation was implemented through the *fallocate* interface. Therewith a file can be physically expanded to a requested size and the expansion is marked as uninitialised. *Fallocate* requires the usage of extents, because only extents support the "uninitialised" label. Reads from the expanded and uninitialised region should return zeroed out data. A write to such a block splits the effected blocks from their extent and put them with new and initialised extents back into the extent tree. Due to the random access character of the hash table and DIA usage, the extent trees are subject to frequent extent splits and inserts which lead to heavy extent tree growths. This solution causes additional and avoidable metadata management overhead.

3.2.2 Zero out en bloc

We decided to allocate and to physically zero out all blocks en bloc at the moment of their first access. This results in a delay of the first access' response time but also results in a very flat extent tree and no further tree modifications and thus no later delays due to initialisation.

3.3 Effects of hash table size and occupancy rate

In this section we discus the effects of the hash table occupancy rate to the filesystem performance. The number of available hash table slots determines the maximum occupation rate. Therefore we describe the possible hash table dimensions at first. Afterwards, we regard the performance implications.

To enable a content/location mapping ext4fs+dedup uses a hash table, whose size corresponds to the number of filesystem blocks. The minimal number of

required slots to handle the worst case of a filesystem, which is completely filled with a disjoint data set, then every slot is occupied.

The occupancy rate of the on-disk hash table has the same performance effects as known in the context of in-memory hash tables. A heavily occupied table tends to more collisions and thus an increased probing effort.

However, one could also use a larger hash table, that never gets fully occupied. The usage of a power-of-two-sized and block aligned hash table would also ease the derivation of the home index out of the fingerprint, because one only would need to take $ld(hash\ table\ size)$ bits of the fingerprint (a simple bit shift) instead of a more expensive modulo operation.

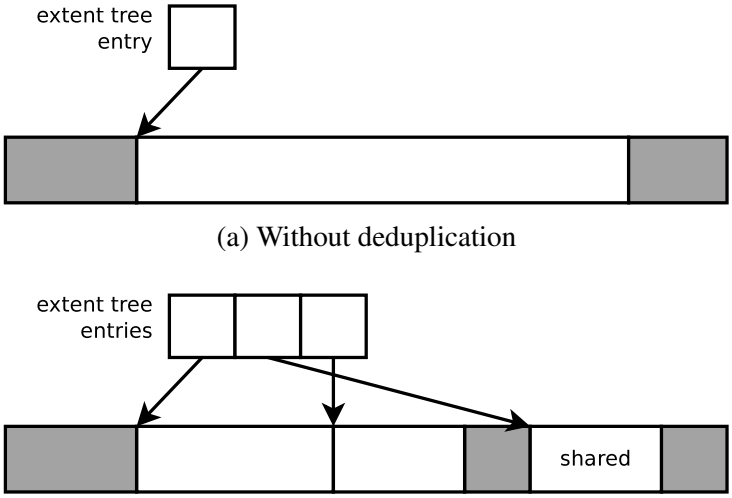
The power-of-two-sized hash table would cover n entries where n is the next power of two greater than or equal to the number of blocks in the filesystem. Thus the case with the smallest possible maximum occupation rate is met, when the filesystem has $n/2 + 1$ blocks. Then this hash table would nearly occupy twice the amount of storage that the unaligned variant does. Thus the occupancy rate of the hash table is about 0.5 at most. Whereas the occupancy rate of the unaligned hash table exactly corresponds to filesystem's capacity utilisation.

Due to the variable maximum occupancy rate of the power-of-two hash table design, such implementations would show significantly varying performances in terms of access time depending on the number of filesystem blocks.

3.4 Drawbacks of deduplication with extents

In the context of deduplication with block granularity, the extent feature looses attractiveness, because deduplication inherently tends to filesystem fragmentation and thus to smaller contiguous block areas.

If a larger contiguous number of blocks is written to disk and we detect a duplicate block in the middle of that, we will break the extent into two pieces (the one before and the one after the duplicate block) resulting into a file consisting of three extents (part before, duplicate, part after) instead of one without deduplication (cf. Figure 3.6). Therefore deduplication as an intensifier of fragmentation leads to larger extent trees. But the saved data blocks should compensate the increased meta data.



(a) Without deduplication

(b) With deduplication

Figure 3.6: Deduplication-inherent fragmentation causes a growth of the extent tree.

Chapter 4

Evaluation

In this chapter we evaluate the performance of ext4fs+dedup. First we describe the used evaluation environment and the benchmarks, then we present and interpret the benchmarking results.

4.1 Evaluation environment and benchmark data sets

The following evaluation has been done on commodity PC. To ease development and evaluation, the modified Linux kernel was tested and evaluated on a User Mode Linux (UML) instance. UML ran Debian 6.0 Linux hosted on a Dell Latitude E6500 notebook with an Intel dual core processor and 4 GB of main memory. Therefrom 1 GB was assigned to the UML instance. A Samsung SSD 830 Series solid state drive connected to the notebook over eSATA was formated with our duplicate aware filesystem ext4fs+dedup. All samples taken are compared to an unmodified ext4fs (as published with the 3.4.6 Linux kernel) run in the same environment. The examined filesystems had a size of 8 GB.

Deduplication mainly effects filesystem writes. We have tested the implementation in extreme conditions, with disjoint data sets on the one hand and redundant data sets on the other hand, as well as in everyday copy operations and in the filesystem benchmark `bonnie++`.

Corner cases To reliably create disjoint and redundant data sets respectively we wrote a small programme ourselves. In the case of redundant data sets, every file consists of 1024 identical blocks. In the case of disjoint data sets, every block is unique in the filesystem. Both benchmarks write 1024 files of 1024 blocks each, resulting into a total data set size of 4 GB (with 4 KB filesystem blocks).

Copy benchmark This benchmark evaluates the performance of ext4fs+dedup facing everday data sets. For the sake of confirmability we selected the public DVD image of openSUSE 12.1 (64 bit version). It is 4.6 GB in size. We copy this image once within each tested filesystem in this benchmark.

bonnie++ `Bonnie++` offers two different benchmarks. One simulates random read, write and rewrite operations in random access as commonly seen on filesystems hosting databases. The other one focuses on stressing the filesystem implementation with directory modifications with the creation and removal of many files

The first benchmark is of interest, because it heavily uses the remapping functionality triggered after a duplicate detection or after the overwriting of a shared block (copy-on-write). The second benchmark is of no avail, for limiting modifications to directories does not benefit general throughput benchmarking.

Therefore we confined us to the first `bonnie++` benchmark. We make one benchmark pass with an effective file size of 4 GB and use the fast mode, which skips the operations of character granularity. This is the resulting configuration: `-x 1 -s 4g -f -n 0`. See Table 4.1 for descriptions of the command line flags.

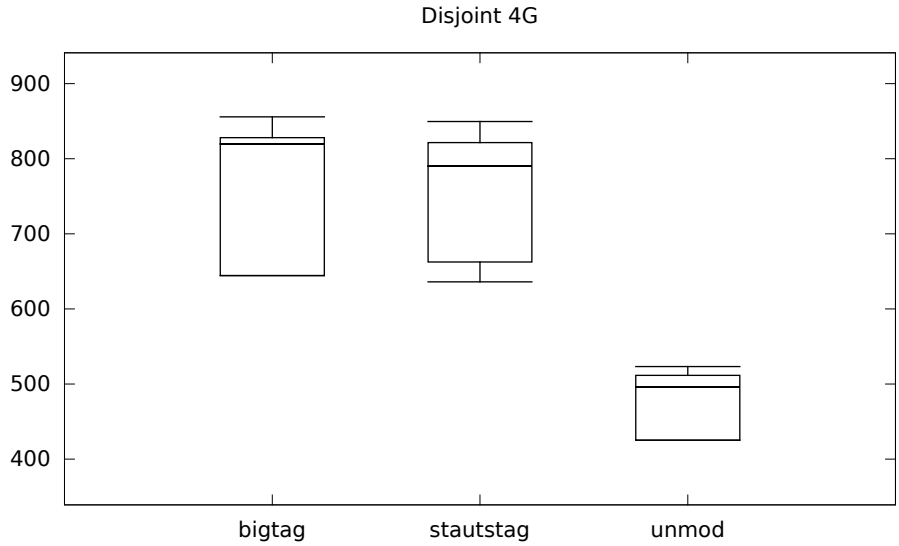
flag	meaning
x	number of testing passes
s	size of file used for first benchmark
f	enables fast mode avoiding character granular operations
n	number of files created in the second benchmark (zero disables)

Table 4.1: This table explains the meaning of `bonnie++`'s command line flags.

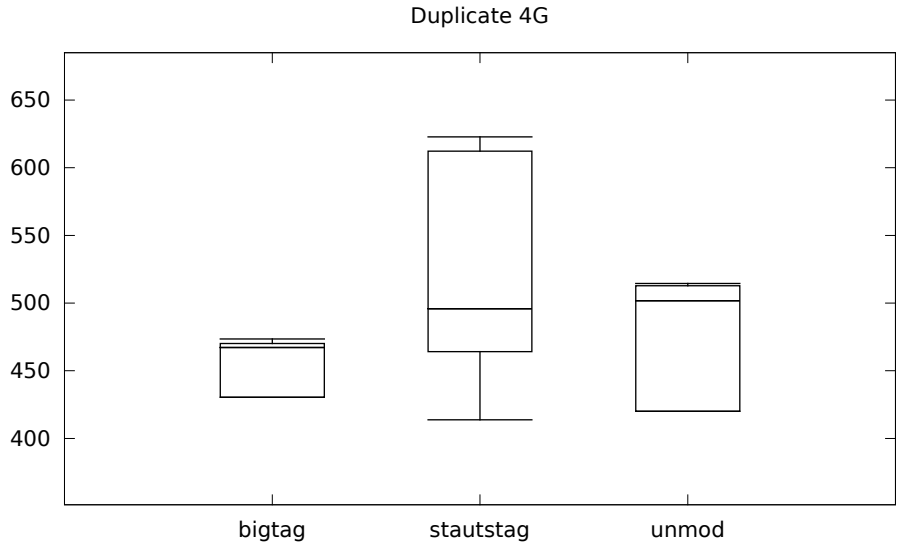
4.2 Corner cases

Figure 4.1 illustrates the results of the corner case benchmarks.

The benchmark that writes only disjoint blocks took 754 ± 118 seconds in the status tag variation and 488 ± 63 seconds using the unmodified ext4fs. For the other benchmark that writes duplicates, we measured 520 ± 106 for status tag and 489 ± 69 for unmodified ext4fs. This results into a deduplication overhead of 55 percent for the disjoint write and about 7 percent for the duplicate write.



(a) Disjoint data set



(b) Duplicate data set

Figure 4.1: Results of the corner case benchmarks. The boxplots show the duration of the benchmarks in seconds.

4.3 Copy benchmark

Figure 4.2 shows the duration of copying the image for the unmodified ext4fs and the two different tag variations of ext4fs+dedup.

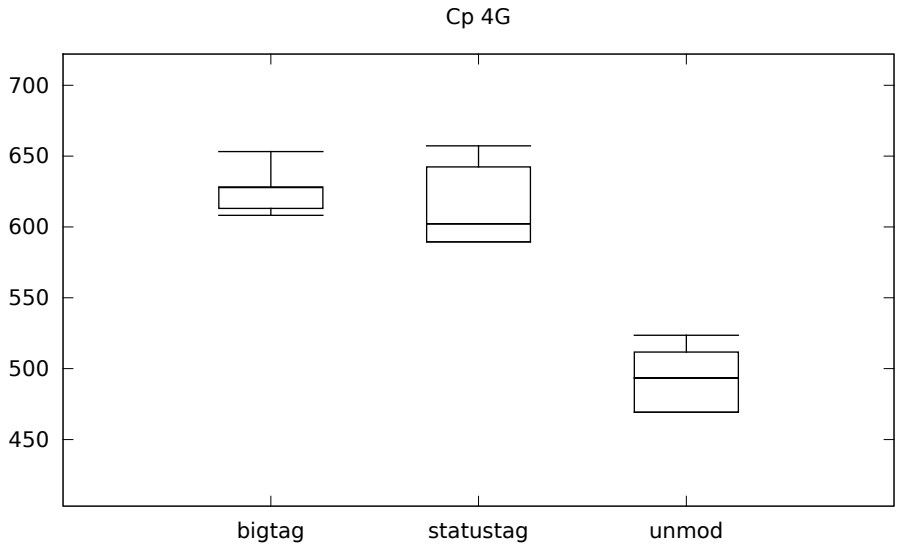


Figure 4.2: Results of the `cp` copy benchmark. The boxplots show the duration of the copy in seconds.

The results for the two tested tag variations barely differ. Big tag shows an average copy duration of 626 ± 17 seconds, compared to status tag with 618 ± 28 seconds. The unmodified ext4fs needed 495 ± 26 seconds to write the test image. This implies deduplication overhead of about 25 percent compared to the baseline.

4.4 Bonnie++

Figure 4.3 shows the results of the `bonnie++` benchmark. As in the copy benchmark, the difference between the two tag variations is marginal. The average duration is 2967 ± 238 seconds for big tag and 2702 ± 194 seconds for status tag. Benchmarking the unmodified version resulted into a duration of 433 ± 31 seconds. Thereby the overhead of deduplication is about 530 percent. Writing takes more than six times longer with deduplication.

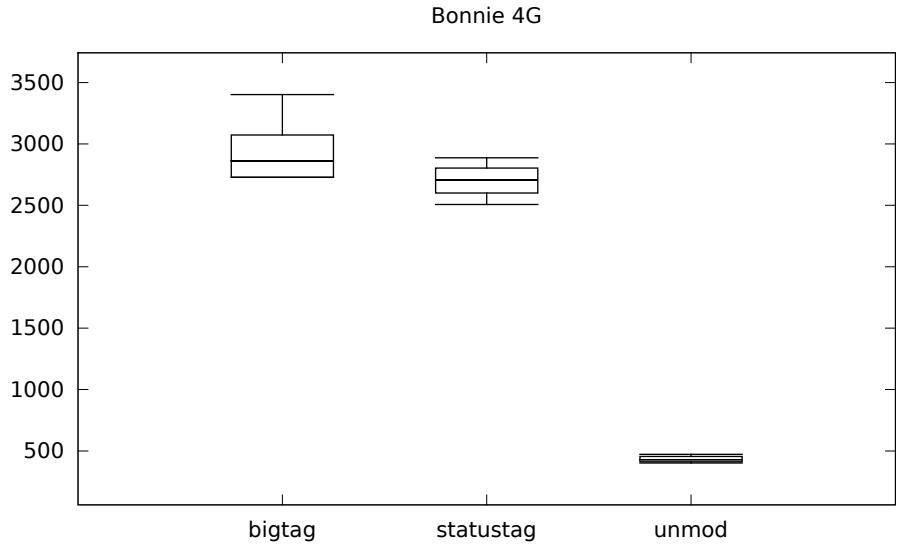


Figure 4.3: Results of the `bonnie++` benchmark. The boxplots show the duration of the benchmark in seconds.

4.5 Conclusions drawn from benchmarks

4.5.1 Tag variations

We expected the big tag variation to be at least slightly faster than the status tag variation, but the opposite came out. Indeed, the differences are small, but status tag was faster in every pass. The theoretical reduction of full fingerprint comparisons seems not to compensate the greater tag comparison effort.

Perhaps the larger record size contribute to this worse performance. Due to the additional bytes for the big tag, less records fit into a filesystem block and thus more block fetches may be necessary.

4.5.2 Deduplication overhead

According to our expectations, the duplication overhead vitally depends on the treated data set. The duplicate write benchmark could nearly compensate the deduplication overhead—caused by the fingerprint calculation and reference counter management—with the saving of the actually repeated data write.

The other extreme of writing totally disjoint blocks maximises the deduplication overhead. In contrast to the duplicate extreme, the arising costs for fingerprint calculation and hash table and DIA insertions cannot be compensated at all, because the data block still has to be written.

Between these two extremes, the DVD image is classed. Ext4fs+dedup detected up to 130 duplicate blocks depending on the moments of flushing. Therefore, you can state the rule of thumb: the more and the larger duplicates there are, the faster the data set is written out.

Regarding the extremely bad performance of the `bonnie++` benchmark on ext4fs+dedup, it is the result of the very frequent modifications. First, `bonnie++` write a lot of identical blocks, which it partially rewrites next, which causes several copy-on-write processes, whereas the unmodified ext4fs can easily overwrite the to-be-rewritten blocks without any further allocation or mapping effort.

Chapter 5

Related Work

In this chapter we present publications proposing or describing implementations to deduplicate data in background storage. Based on the categorisation of Srinivasan et al [11] we subdivide existing deduplication approaches according to their primary optimisation objectives. According to Srinivasan et al, the subdivision distinguishes between inline and offline as well as latency sensitive and throughput sensitive systems (cf. Table 5.1).

In the following we first define the mentioned distinctive features. Afterwards, we focus on inline deduplication systems, refine the subdivision accordingly, propose a categorisation, name exemplary implementations and compare them to our approach. Finally, we regard offline deduplication and show collaboration posibilities with inline systems.

	inline systems	offline systems
latency focused	iDedup [11], dedupfs [4]	NetApp [1]
throughput focused	ChunkStash [5], EMC Data Domain [14], Deep Store [13], Hydrastor [6], Sparse indexing [8], Symantec [7], Venti [10]	n/a (seems to be motivation for that)
general purpose	ext4fs+dedup, ZFS [3]	Btrfs [2]

Table 5.1: Tabular overview of related work.

5.1 Distinctive features

The major distinctive features are the deduplication point in time and the performance objective.

Deduplication time The deduplication either takes place during the data flush to the storage—thus inline with the write path—or during idle times of the storage system—thus offline to the write path. The latter approach requires and retroactive deduplication.

Deduplication performance Depending on the use of storage system, shot latency requirements outweigh high throughput requirements or vice versa. Primary storage systems host frequently accessed application data (e.g. databases), which commonly serve latency sensitive remote requests. Therefore latency reduction optimises performance. Secondary storage systems are commonly backup or archive (primary) data during idle times (e.g. over night). Therefore, throughput is critical with respect to a timely completion.

5.2 Categorisation of inline deduplication systems

Focusing on inline deduplication systems we refine the above distinction by taking two additional aspects into account: deduplication completeness and memory consumption. Including the already regarded latency aspect, the following three characteristics determine the behaviour of an inline deduplication system:

- Degree of deduplication completeness: This is the ratio of typically detected to actually present duplicates.
- Latency: This is the delay caused by duplicate detection, that is added to the write path.
- Memory consumption: This is the amount of main memory, that the deduplication system uses for duplicate detection related mechanisms (e.g. fingerprint index).

These three properties are interdependent. One can only optimise two properties while the third has to be subordinated. For instance optimising the deduplication completeness and the latency, requires a fast accessible and complete index of all stored data, thus consuming a lot of main memory. This optimisation interdependency results in three coarse categories of inline deduplication systems, which is illustrated in Figure 5.1. We call this the “inline optimisation triad”.

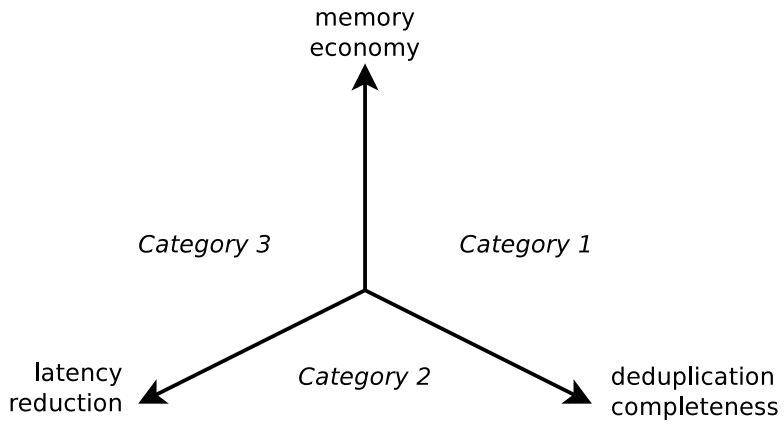


Figure 5.1: Inline deduplication systems subdivide into three categories depending on which two properties are optimised at the expense of the third.

5.2.1 Category 1: Subordinating latency

This category comprises deduplication systems, which optimise deduplication completeness and main memory economy, subordinating the latency aspect. Such systems typically fulfil backup purposes with high throughput requirements and huge datasets making in-memory deduplication indexes economically infeasible. ChunkStash [5] locates its deduplication index on a flash drive as a compromise between expensive DRAM and slow spinning disks. Zhu et al [14] describes the employment of a Bloom filter to mitigate the duplicate lookup delay by avoiding unavailing on-disk index lookups of not present fingerprints.

Bloom filtering can be a beneficial complement to ext4fs+dedup, but as it is a kind of deduplication index cache, it contradicts our design objective not to use dedicated caches.

5.2.2 Category 2: Subordinating memory economy

Deduplication systems in this category have to consume lots of main memory to provide a high degree of deduplication at a minimal delay. The enormous memory requirements either limit the system's size or the latency goals. For example, ZFS [3] has introduced a flash-based extension to its in-memory deduplication index to make the deduplication of growing filesystems affordable at the expense of the latency.

In-memory indices are inherently incompatible to our approach.

5.2.3 Category 3: Subordinating deduplication completeness

Within this category deduplication systems focus on latency reduction and economical main memory usage. Hence these systems typically maintain caches containing a partial deduplication index. Dedupfs [4] and iDedup [11] use fixed-size LRU caches comprising the recently observed deduplication metadata. Therefore, the duplicate detection is not complete, as only duplicates within temporally local writes are recognised. Both systems provide configurable cache sizes, which allows tradeoff between memory consumption and deduplication completeness. Furthermore, iDedup reduces fragmentation by introducing a threshold length of deduplication-worthy duplicate sequences. Sequences that don't exceed the threshold are written and not deduplicated.

Again, in-memory caches contradict our objectives. A duplicate length threshold is possible add-on to ext4fs+dedup to mitigate fragmentation at the expense of completeness.

5.3 Offline deduplication systems

Deduplicating offline to the write path neither effects latency—excepting increased read latency due to fragmentation—nor throughput, provided that sufficient idle periods are present to not interfere with productive workloads. Exemplary systems are [1, 2].

Offline mechanisms can complement inline systems, which don't detect and deduplicate every redundancy. Ext4fs+dedup can benefit by offline deduplication in order to remove simultaneously flushed duplicates or deduplicate already written data after belated activation of the inline deduplication feature.

Chapter 6

Conclusion

The aim of this thesis was to evaluate the feasibility of inline filesystem deduplication without main memory caches. Therefore we developed ext4fs+dedup, an extension to ext4fs, which totally abandons any deduplication-dedicated memory cache. Based on this system we ran several benchmarks on a recent Solid State Drive. The measured overhead ranges from 7 to 55 percent depending on the data set.

This performance may be acceptable for general purpose setups, which don't require throughput optimisations to, for instance, complete within narrow backup windows. Decreasing prices, will lead to a wider usage of Solid State Disks in PCs and commodity storage appliances. Therefore cache abandoning deduplication systems will be of particular interest for private use and small organisations, which have no dedicated storage and backup systems. They commonly integrate several services on one hardware and thus will benefit from the low memory footprint of our cache-abandoning approach, because the less memory is required for deduplication, the more is available for other services.

6.1 Future work

To prove practicability of the cache abandoning approach, ext4fs+dedup should be tested in real world storage environments, whose timing and throughput requirements are less strict and which thus may tolerate up to 50 percent overhead.

Our implementation, as presented with this thesis, is confined to the very basic features of ext4fs in order to provide a prototype for evaluation. In terms of deployability, the support of the old indirect block mapping method should be added as well as the support for ext4fs' different journaling modes and its resize and downgrade capabilities.

Resizing the filesystem requires a reordering of the hash table (HT) as the

index derivation of the home slot depends on the filesystem size. To reorder, one could lookup the full fingerprint for each valid record in the old HT, calculate the new home slot and copy the record to the new HT. Afterwards one could discard the old HT and use the new one.

Bibliography

[1] C. Alvarez. Netapp deduplication for fas and v-series deployment and implementation guide. Technical report, Technical Report TR-3505, NetApp, 2011.

[2] Josef Bacik. Offline deduplication for btrfs. `http://article.gmane.org/gmane.comp.file-systems.btrfs/8448`, 2011.

[3] Jeff Bronwick. Zfs deduplication. `https://blogs.oracle.com/bonwick/entry/zfs_dedup`, 2009.

[4] Aaron Brown and Kris Kosmatka. Block-level inline data deduplication in ext3, 2010.

[5] 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, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.

[6] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *Proccedings of the 7th conference on File and storage technologies*, pages 197–210. USENIX Association, 2009.

[7] F. Guo and P. Efstathopoulos. Building a highperformance deduplication system. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 25–25. USENIX Association, 2011.

[8] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies*, pages 111–123, 2009.

[9] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings*

of the 2007 Ottawa Linux Symposium (OLS 2007), volume 2, pages 21–34, Ottawa, Canada, June 2007.

[10] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, volume 4, 2002.

[11] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[12] X. Wang, Y. Yin, and H. Yu. Finding collisions in the full sha-1. In *Advances in Cryptology–CRYPTO 2005*, pages 17–36. Springer, 2005.

[13] L.L. You, K.T. Pollack, and D.D.E. Long. Deep store: An archival storage system architecture. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 804–815. IEEE, 2005.

[14] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.