Harnessing Metadata Characteristics for Efficient Deduplication in Distributed Storage Systems

by

Matthew Goldstein

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2011

Author .....
Department of Electrical Engineering and Computer Science
November 8, 2010

Certified by .....
Robert T. Morris
Professor
Thesis Supervisor

Certified by .....
Jiri Schindler
NetApp ATG
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Harnessing Metadata Characteristics for Efficient Deduplication in Distributed Storage Systems

by

Matthew Goldstein

Submitted to the Department of Electrical Engineering and Computer Science
on November 8, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

As storage capacity requirements grow, storage systems are becoming distributed, and that distribution poses a challenge for space savings processes. In this thesis, I design and implement a mechanism for storing only a single instance of duplicated data within a distributed storage system which selectively performs deduplication across each of the independent computers, known as nodes, used for storage. This involves analyzing the contents of each node for objects with characteristics more likely to have duplicates elsewhere, particularly using duplication within a node as the indicative property - an object duplicated many times in a dataset will likely be duplicated at least once in some node. An inter-node system is responsible for efficiently collecting and distributing the information of these potential duplicates. A test implementation was built and run on several data sets characteristic of common storage workloads where deduplication is important, while distributing across 128 nodes. The efficiency of this implementation was analyzed and compared against the savings when distributed across 128 nodes with deduplication performed only between duplicate objects within each node, without inter-node deduplication. The inter-node deduplication was found to increase the space saved by a factor of four to six times. This represented in one case, a file storage server, only a quarter of potential savings due to the majority of potential savings being in files with only a few duplicates. This left a low probability of locating duplication within a single node. However in another case, a collection of over 100 virtual machine images, nearly all potential duplicates were found due to the high number of duplicates for each object, providing an example of where this inter-node mechanism can be most useful.

Thesis Supervisor: Robert T. Morris
Title: Professor

Thesis Supervisor: Jiri Schindler
Title: NetApp ATG

Acknowledgments

I would like to thank my parents for their love and support all my life and for raising me to have such a strong appreciation for curiosity and learning. I would also like to thank my friends for keeping me happy and helping me to enjoy life even while buried deeply working on this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deduplication is a technique used to reduce the amount of disk storage consumed by content which is duplicated in multiple objects within a storage system. One common approach to finding duplicated content is storing a cryptographic hash of every object such that each object's hash can be compared against those of every other object in the system to find duplicates. When a duplicate is found, one copy of data is deleted and replaced with a pointer to the other copy, saving space. While this has been done easily in centralized storage systems, storage systems have grown and may now be distributed over many independent nodes. This thesis is motivated by the issues encountered when trying to perform deduplication in a distributed storage system. These challenges come primarily from the fact that information is not fully shared between individual nodes. Also, data distribution policies, which determine where any given object is placed, are not congruent with data deduplication.

I propose using existing object characteristics to identify objects that are likely duplicated across independent storage nodes rather than just within each individual node. A system leveraging metadata already present for each object in the system makes deduplication on the node level possible. I contend that it has little performance or disk usage impact on any given component of the system but identifies many of the duplicate objects. Restricting comparisons across nodes to the limited scope of only objects identified at the node level makes locating duplicates across all nodes of the system much more practical. I will use duplication within a node as

the metadata characteristic, or "hint", for the experiment of this thesis. The use of this characteristic for hinting is based on the supposition that if an object is duplicated many times within a dataset, there should be at least one node in which it is duplicated at least once.

The main contribution of this thesis is a method to efficiently analyze individual nodes of a distributed object storage system for content duplicated across nodes. While deduplication could easily be done via a brute force method, the challenge lies in doing deduplication in a manner which scales up with the number of nodes without disrupting normal storage operations or consuming excessive disk space for extra metadata. This method is managed in a manner which makes no assumptions about the distribution of objects across nodes in a content-related context, functioning efficiently despite a random distribution method.

The prototype system produced for the experiments of this thesis operates using duplication within a node as the metadata characteristic, or "hint", for attempting to find duplicates of an object on other nodes. The experimental system, working on datasets distributed across 128 nodes, was compared with the savings on those same distributed datasets by a mechanism eliminating duplicates only within each node based on object hash, without inter-node deduplication. While such a mechanism would find all potential duplicate objects if the dataset were stored on only one node, distribution across all nodes makes it much less effective. The experimental system was able to improve duplication space savings by a factor of 4 to 6 times. With a dataset of many virtual machine images, this was nearly all potential duplication due to the high number of duplicates of any given object. Cases such as this, with high levels of duplication of objects, are where the cross-node deduplication mechanism developed in this thesis is most effective. However, with a file storage dataset, the improved savings still left the majority of potential savings unobtained; due to the large portion of potential savings comes from files with few duplicates. This meant that for many files no duplicates were located within any given node to trigger the cross-node deduplication.

An attempt was made to use hinting based on individual block duplication in-

formation. However, I found that this was excessively computationally and memory intensive. I also learned through this additional experimentation that hints at the whole object level are much easier to deal with. This is because using a block hint for moving objects leaves a hard problem of determining which objects to move to which node in order to gain the most savings.

This work applies to share-nothing systems which distribute objects across independent storage nodes. In this context, there is not full sharing of information between nodes, each node keeping object metadata including size and hashes to itself. An object can be a file, and extent of a larger entity such as a large networked file representation, or a binary large object. I examine single instancing between nodes on both the object and 4KB block level. I also make use of existing internal deduplication within each node.

1.1 Deduplication

Deduplication is a form of compression whereby multiple instances of the same byte sequence are removed and replaced with pointers to the location of the remaining one within the system. This deduplication can be applied with a granularity of objects such as entire files or individual blocks. File duplication happens for several different reasons, depending on the source of the data. User files are sometimes duplicated when a single user wants to make a backup of data which they are about to edit or when multiple users are working on different copies of similar projects. Similarly, duplicate files can appear when multiple people in the same organization are sent the same file, such as a company report, to review. System files can also be duplicated when software is built for multiple systems, with a full set of files kept for each system. In an environment with virtualized computers and servers, system files can be duplicated between images of computers with similar operating systems and configurations.

When the storage system detects that it is writing data which it has written before, it makes a reference to the original data rather than duplicating it. Because

deduplication is done at a storage system level with global context, the domain of data being compressed is much larger than most other forms of compression, giving the greatest opportunity for finding duplication. Also, because it is built into the storage system, users do not have to specify each set of objects they wish to compress. Reading deduplicated objects also does not need to use the extra CPU time that other forms of decompression require. The processing overhead comes from finding duplicates when the object is written. The duplicated data exists normally on the disk and is read as quickly as any other data.

Deduplication can operate on several levels of granularity. Windows 2000 [1] does full-file deduplication, storing a hash of the entire file. When a new file's hash is calculated, it is compared against the existing hashes and if there is a match the two files are compared byte-for-byte. If the files are duplicates of each other, the new file is not written to disk but a reference made to the existing file. Other systems, such as Venti [6] or ASIS [4], operate similarly, but on a per-block level, keeping a hash of each block on disk. When duplicates are found, a reference is made to the existing block.

While some systems such as Venti and HYDRAstor [3] operate on data as it enters the storage system, other systems, such as ASIS and the one considered in this thesis, work offline and locate recently written objects which are duplicates of existing ones. In this case, the new object is referenced to the old object and the now-unused duplicate data is marked for deletion. In all of these cases, deduplication operates by looking at some unit of storage and determining that two of these items are the same. In this thesis, I look at each object as a whole in the object storage system. Each object may have a different size, but none span multiple storage nodes.

1.2 Distributed Filesystems

As storage needs have grown, the model of large storage systems has moved towards greater distribution. Rather than storing all objects on centrally controlled disks, storage can be distributed across many devices, each one an independent storage

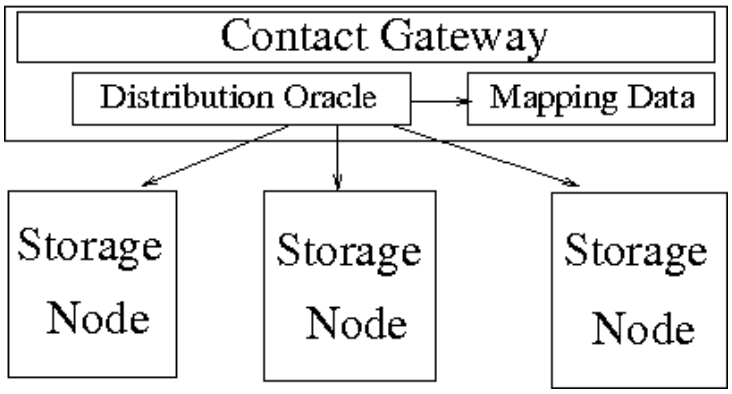

Figure 1-1: A generic distributed filesystem. Objects are presented at the contact gateway. An oracle then distributes the objects to storage nodes and stores the object to node mapping for later use.

node. This allows storage systems to easily increase capacity through the addition of more nodes. Adding nodes improves access speeds by distributing the workload across many nodes.

Several components are involved in the storage and recall of objects in such a distributed system. Generically, as seen in Figure 1-1, objects are passed to the system from a client at some gateway. The gateway accepts the object, and an oracle determines some node in the system where it should be stored. The object is passed to the storage node. A mapping of the object's ID to storage node is given to some metadata storage system. When a client requests a given object, the metadata storage system is queried to determine which object storage node to contact. The object is then retrieved from that node and passed back to the client.

The policy for object distribution across nodes varies by system. In some systems, including Ceph [7] and HYDRAstor there is a consistent distribution algorithm based on the object's hash. In others, objects are placed in the node with the greatest free space, least load, or any other number of factors. Obviously, a distribution based on content such as Ceph's would make locating duplicate objects extremely simple, but such systems are choosing this optimization at the expense of performance. In this thesis, I assume some mechanism which distributes objects randomly across nodes without respect to content. I refer to this component of a system as the distribution

oracle.

1.3 Design Outline

Performing deduplication in the node-based filesystem has two related components. The first is a method to deduplicate objects within a given node against each other in a manner that is not prohibitively processing intensive. The second, related, component is a method for determining potential deduplication between objects stored in different nodes. This assumes the data has already been distributed across nodes in some manner which is not deduplication-driven. I analyze object-level duplication and determine which characteristics may help identify objects for deduplication. I implement these characteristics in my system.

Intra-node deduplication is largely a solved problem; deduplication based on hashes of objects or the common fixed-size or variable-size blocks, is available in several commercial systems including NetApp ONTAP's ASIS and DataDomain [9]. Hash based deduplication can be done by keeping a lookup table of all hashes of objects within a given node. The storage overhead from the use of hash tables may be reducible by using the hinting between nodes which I will be developing, but if not it should not have to be too large because each node need only store the hashes for its own objects so no global list is stored.

Determining potential deduplication between objects stored in different nodes is harder than the intra-node case. I propose that while an exhaustive search for all duplicates across the entire distributed system is cost-prohibitive, finding most duplicates efficiently, without a giant central hash table should be possible with hints based on metadata already available in the system. This depends on a dataset with high levels of duplication, such that some pattern can be discerned while looking at only a small portion of the data. I have developed a model of and prototyped such a storage system using a metadata-dependent methods to perform deduplication hinting and determine the effectiveness of this method.

1.4 Thesis Overview

The thesis is organized as follows. Chapter 2 discusses the motivation for this thesis, why traditional approaches are impractical, what challenges this system faces. Chapter 3 details the design of my system, including the duplication hinting characteristic and the processes used to locate and move duplicates. Chapter 4 outlines the procedure I followed for building the system and the tests I ran. Chapter 5 is an analysis of the results of running this system with several test cases. Chapter 6 contains the conclusions of this thesis and Chapter 7 is a review of related works.

Chapter 2

Motivation

This chapter examines the differences between a distributed storage system and a centralized storage system, which motivates this thesis. This includes the scaling of duplication detection in terms of both computation and memory usage. I then present potential solutions and explain the conditions required for them to work.

2.1 Multiple nodes

Large distributed systems make traditional deduplication with content-based hashes complex. Each node in a distributed system is responsible for the low-level details of how it stores and references an object. As independent share-nothing nodes, each node keeps these storage structures internally and is not made aware of the data in other nodes. As deduplication is performed, that node manages reference counts to each object to ensure that no object is removed if it is still referenced elsewhere. This would be made much more difficult if there were also global references due to deduplication across nodes which must be accounted for, so deduplication can only occur within each node.

Deduplication was traditionally performed from a top level in previous storage systems rather than delegating the responsibility to each node as in my system. However, a table tracking the hashes of all objects across a storage system of multiple Peta bytes in size would be too expensive to keep in memory, particularly when it is

continuously updated with new entries.

2.2 Distribution Policies

By distributing objects across nodes in a content-based manner, the problem of multiple nodes could be easily solved. With a distribution oracle sorting objects by content, as implemented by Ceph and Hydrastor [3], the task of identifying duplicates is made simple as all duplicates of an object would already be on the same node. However, sorting by content requires calculating some hash of the content which produces computational overhead. This overhead is incurred by the storage system, or by clients, requiring clients to be written for this particular storage system. Further, such a content-based distribution oracle would not take into account the level of service required for a particular object if the object storage system provides a service level agreement, or SLA, which could determine the amount of replication. Similarly, this method does not allow smart distribution of objects across different nodes or on the same node for more rapid sequential access depending on disk configuration.

Attributes besides hashes could be analyzed and global structures built to support deduplication. However, in a system focused on performance, with deduplication as a secondary consideration, complex structures which may impact performance cannot be used in the storage pipeline. So, this system has the constraints of running purely offline on each node in a manner that generates little to no metadata of its own in order to minimize performance impact.

Chapter 3

Design

This chapter outlines the principles and mechanisms used in this thesis to locate duplicates across nodes. The term "hint" refers to a metadata characteristic indicative of duplication, which this deduplication system uses as the basis for checking if a particular object is duplicated in other nodes. The metadata structures used to support deduplication are also discussed.

3.1 Reference Count Hinting

When looking for duplication across nodes in a storage system, any duplication of an object within a single node may provide a hint that the object is duplicated elsewhere as well. Consider a storage system with 100 nodes. Even with such a large storage system, there is a good chance that similar objects will be written to a given node. This is similar to the "Birthday Problem" [8] but with fewer dates. Even with randomized object distribution, it would take only 13 copies of an object to have a 50% chance of a collision within some node, and 95% at 24 copies. This thesis considers random distribution because a uniform distribution of duplicates across nodes will result in the fewest duplicates within a single node. A real distribution mechanism may operate by file directory or some other characteristic, likely resulting in more duplicates being located. As detailed in section 4.1, high levels of duplication are common in large datasets.

If a node is already performing internal deduplication, it typically already stores a reference count for any given object or block, whichever the node is deduplicating. These counts are necessary when performing deduplication to ensure that if data is changed or deleted there is no accidental deletion of other objects which previously had identical data. Scanning a node for any objects with reference counts greater than one will collect a set of potentially further-duplicated objects.

Reference count is the primary characteristic of cross-node hinting used in this thesis. As corroborated by my experimental results, it works well for any dataset with a high portion of possible duplication savings coming from objects with high duplication count. Even if a dataset has a majority of unique objects with only a few duplicates, it is possible for the high level of duplication in those objects with many duplicates to offset this and account for more of the space savings. For example, in a company with 100 employees while each may have a personal file that they have duplicated and may not be found duplicated within a single node it only takes one report sent to the entire company, and thus more likely to be duplicated and found withing a single node, to account for just as much duplication savings. While reference count may not be comprehensive in locating duplicates across nodes, it should find some high percentage of them. The chances of finding duplicates this way increases as the dataset being deduplicated has more duplication, making this characteristic more useful in datasets with greater potential deduplication space savings.

Several other hinting mechanisms were considered but not used in the thesis experiment. Service Level or SLA, such as the amount of redundancy for a particular storage volume, and size both could serve to provide hints without requiring the extra hash calculations. However, neither can completely identify potential duplicates on its own and would only be useful as a compliment to a more complicated hinting scheme. File-based attributes, looking for similar file names or directory structure, were also considered but not used in this thesis because they do not apply to a general, non file-based object storage system.

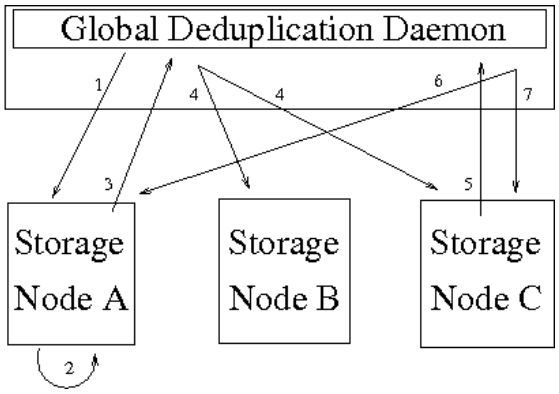

Figure 3-1: The Deduplication Process. 1: The Global Deduplcation Daemon asks each storage node for a set of potentially duplicated objects. 2: The node-level deduplication scanner locates any internal duplicates to report as interesting. 3: The node returns a list of interesting objects. 4: The global daemon asks each other node if it has any of these same objects. 5: Node C reports that it also has one of the interesting objects. 6: The global daemon stores the duplicate object from node C on node A where it can be internally deduplicated against the existing identical copy. 7: The global daemon asks node C to delete its copy of the duplicate.

3.2 Deduplication Process

Deduplication across nodes, as shown in Figure 3-1, begins with the system's global deduplication daemon requesting that each node compile a list of objects which have characteristics within the node suggesting that the object may be duplicated across nodes. In a running storage system, scanning for such objects may be done in a variety of ways such as at write time, as a periodic local background activity on each node, or as suggested earlier, as a response to an explicit request from the global daemon. The method of identifying objects based on their duplication within the node is performed for each object by a simple scan across the metadata of all objects in the node. The node scanner adds suitable objects to the list of interesting objects to be reported to the system-wide deduplication daemon.

Next, the global deduplication daemon compiles these per-node lists of interesting objects into one list and queries each node to determine if that node also has any of these objects. The node responds with a list of the intersection of the set of objects it was asked about and the set of objects it stores, thereby identifying cross-

node duplicates. The duplication daemon then moves duplicate objects between the nodes, bringing more duplicates into the same node and allowing for greater space savings through intra-node deduplication.

Each of these steps requires a decision to determine what is reported and moved where. I will refer to each decision-maker as an oracle. The first decision oracle is each node's initial reporting of interesting objects. While the experiment was run identifying duplicate objects within a node as the hinting mechanism, it could also easily be used to report duplicate blocks or other characteristics. The second oracle is also at the node level. This oracle decides which of the objects already found to be of interest elsewhere it should report as intersecting. Finally, the global deduplication daemon decides which reported duplicates from the nodes should result in object moves across nodes. In this experiment, the second oracles reports and the third oracle moves all located intersections.

3.3 Metadata Structures

Within each node, there is a scanner process to search the contents of that node for objects which have characteristics indicative of likely duplication with objects stored on other nodes. To support this process, a hash is stored as metadata for each object and as part of the descriptor for each block. This scanner will run on request across each node and keep track of any objects which seem likely to be duplicates.

For inter-node deduplication, periodically, a global daemon will check with each node for any likely duplicates. That node's scanner will report any likely duplicated objects it has found internally, sending the objects' ID and hash. The node also sets a checkpoint so it will not to report the same objects in the future. After the daemon has obtained all likely duplicates across all nodes, it creates a list of unique instances of duplicate objects. It then contacts each node again, asking if it has any of the potential duplicate objects. To make this efficient, each node has already stored a lookup table from hash to object ID. When any duplicates are found, that object's data is removed from all but one node and the mapping of each object with identical

content has its path set to the node containing the single instance of its data.

Chapter 4

Experimental Setup

This chapter describes the data acquisition and test infrastructure developed to perform the experiment of locating duplicated content across nodes in a distributed storage system. This experiment determines if the reference count hinting scheme is able to restore the deduplication savings lost by distributing a dataset across independent nodes. First it analyzes the datasets to find the space savings possible in a centralized storage system by comparing the hashes of all objects. Then it determines the space savings lost when the content is distributed across independent nodes, comparing only the hashes of objects to others in the same node. Finally, it finds the space savings after using the mechanism proposed in this thesis. Three sets of data are used, ranging from 800GB to 1.4TB in size, each of a different origin, representing a different type of workload. In order to perform the same set of experiments across these datasets which were meant to be accessed in different ways, I created a format to which all of the datasets were converted. For each dataset I created an acquisition tool used to perform this conversion. For this experiment I built a distributed storage system, as well as a tool to load the normalized datasets as represented in Appendix A into the storage system.

4.1 Characteristic Datasets

The first of the three datasets is the contents of Makita, an internal NFS/CIFS server at NetApp. Makita is used primarily to store files such as presentations and documents, some of which may be shared outside the company. The storage on this server is used by those in engineering, marketing, sales, and administrative positions. The dataset contains 12.4 million files comprising 1.4TB of data and was obtained directly through a tape backup of the server. Because the Makita dataset is file-based, each object is a file, so there is no uniform per-object size. There is about 36% duplication on the whole-file level, meaning 36% of all files are duplicates of some original file with identical contents, only differing by name or path. These files account for about 24% of the space used in the dataset as most of the duplicated files are smaller than average. The original data was stored by in 4KB filesystem blocks, so block-level duplication was also examined, finding 37% duplication at the individual 4KB block level within the Makita dataset. Interestingly, most of the savings comes from eliminating whole files with identical content.

The second dataset is the contents of an Oracle database server supporting administrative tasks within Netapp. The dataset consists of an 800GB Oracle database obtained directly through a tape backup. Oracle servers use Oracle's Automatic Storage Manager (ASM) to distribute table data across a storage system. ASM works on 1MB chunks of the database at a time. Therefore, 1MB was used as the object size for this dataset. On the object level, there is 13% duplication, while at the 4KB block level, there is 46% duplication.

The third dataset is a 1.3TB image of a VMWare image farm of 113 Windows XP virtual machines for internal use at VMWare. The dataset was collected by reconstructing the image from a series of duplication characteristics in a paper by Clements et al [2] using the tool described in section 4.2.3. VMWare's VMFS handles operations on a 1MB level. Therefore, 1MB was used as the object size for this dataset. On the object level, there is 22% duplication while at the 4KB block level, there is 82% duplication.

Dataset Characteristics			
	Makita	Oracle	VMWare
Size	1.4TB	800GB	1.3TB
Blocks	360,703,391	209,715,200	349,057,626
Objects	12,364,590	819,216	1,363,506
Object Size	Size of File	1MB	1MB
Duplication			
Object	23.9%	13.27%	21.7%
4KB Block	37%	45.95%	82.38%

Table 4.1: General characteristics of the three datasets used in the experiment.

As previously described, duplication space savings can operate on an object or block level. While it is relatively simple to determine the raw amount of space saved through deduplication, I would like to find out where the savings comes from, if there are many blocks with only a few duplicates each or several blocks each duplicated many times. I examined the contribution to total space savings by the number of duplicates per unique object or block. As seen in figure 4-1, the Makita dataset derives most of its savings from blocks with a low number of duplicates, with 50% of savings from blocks duplicated less than 20 times. Little savings comes from individual blocks with many duplicates, with blocks duplicated over 80,000 times accounting for only one percent of savings. Meanwhile the Oracle dataset has most savings from blocks with many duplicates each, with 50% of savings coming from blocks with at least 20 millon duplicates - blocks which were allocated to the Oracle database but not yet used and still filled with all 0s. There are also many with 50 and 100 copies, accounting for 20% of space savings. The VMWare dataset, on the other hand, derives most of its savings from blocks duplicated around 100 times, accounting for 65% of total savings. This matches the dataset's origin as disk images of 113 virtual machines.

Looking at the deduplication in each dataset by object in Figure 4-2, we see a slightly different trend. Makita still has most duplication coming from objects with lower counts, now 77% under 20 duplicates, but nearly all of the Oracle object duplication comes from a single object duplicated 100,000 times. This object seems to consist of all 0s, presumably because the database has not grown to consume the

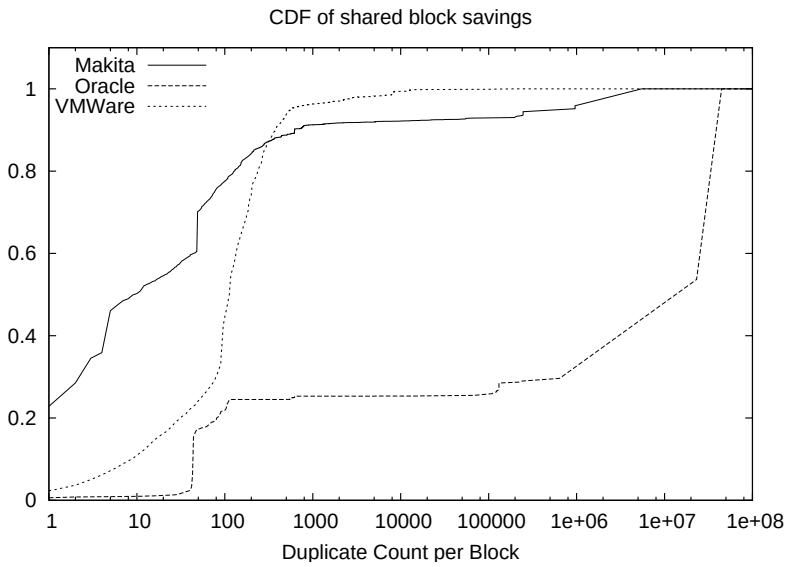

Figure 4-1: CDFs of the number of duplicates contributing to space savings by 4KB block.

entirety of the space allocated to it. The VMWare data graphed here is so different due to the synthetic nature of the data's generation, detailed in section 4.2.3, which cut off object duplication at 120 duplicates.

4.2 Data Aquisition

Data was aquired from several different sources, so I created a cannonical format in order to run the same experiments on all three datasets. This section describes that representation and the tools used to convert each of the datasets from their initial form.

4.2.1 Dataset Representation

In order to perform the same set of experiments across the three different datasets, I converted all three to a canonical format. To run experiments on the datasets in a timely fashion, storing a byte-for-byte representation of all data in each dataset was impractical. Instead, I created a representative format containing only metadata.

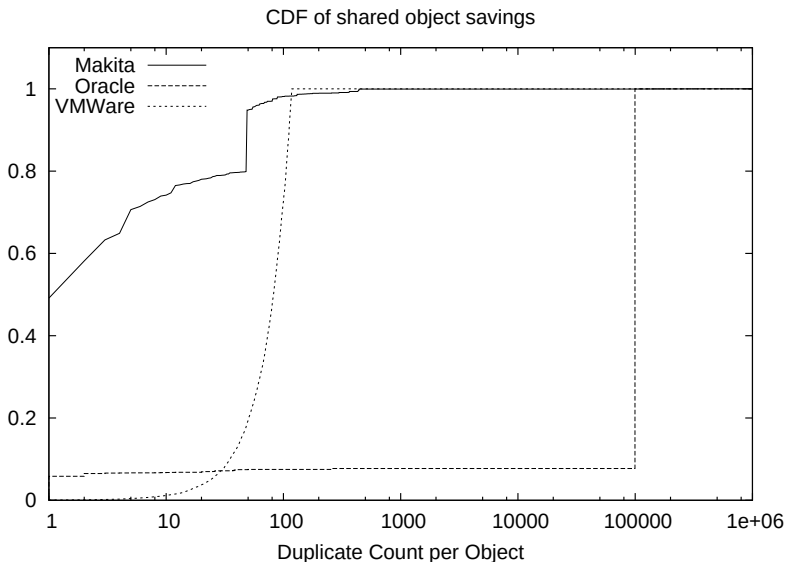

Figure 4-2: CDFs of the number of duplicates contributing to space savings by object.

The metadata, including object and block-level hashes, was extracted and stored in a special file format outlined in Appendix A which completely describes each object in the dataset. The special format takes up only 1% of the space used by the original data. The format consists of a comma-delimited file with each row representing a new object in the dataset. The columns hold the object name, hash, size, filesystem permissions, and then individual 4KB block hashes with each block in its own column. While this means some rows may contain several hundred thousand columns, the format does not require placeholders for these columns in every row.

4.2.2 Direct Generation

Both the Makita and Oracle datasets were generated in the above representation directly from tape backups. To create the Makita dataset, I augmented a tool which scanned all files in that filesystem. For each file, a buffer was filled with the contents of the file. 4KB was read from this buffer at a time and hashed to generate the block hashes for the record. I modified the tools such that for each file, a 128-bit hash was taken of the entire file to be used as the hash for that object record in the representation. The name was also recorded and a system stat call used to obtain the

size and permission bits on the file for its record. The file and block information were combined and outputted in my canonical representation format. This is repeated for all files in the dataset.

Oracle's ASM manages storage of Oracle databases. ASM writes data directly in 1MB chunks rather than as files as part of a filesystem. These chunks can either be written to addresses on an iSCSI object known as a LUN or within a large file. For network-based storage access, the database is presented to ASM as a LUN, to be written to directly, though on the server doing the storage, the LUN may be kept as one or more files. For the Oracle dataset, I obtained the file representation directly from the server which stored the Oracle database LUN. This was performed using special access provided by the debug mode of a non-production FAS system which the LUN was loaded onto from the tape backup. I read 1MB at a time from the file representation of the LUN, generated a hash of that 1MB chunk as the object hash in the representation, and then generated hashes of each 4KB within that object for the block hashes.

For VMWare, data was generated using the tool below from characteristics outlined in section 4.2.3.

4.2.3 Duplication Dataset Generator

For various reasons, full access to large datasets is not always possible for researchers. However, in some cases, broad characteristics of the data, such as the frequency of duplication in the dataset, is available, as was the case for the VMWare dataset. I created a tool which characteristics of a dataset, recreates a representation of that dataset with the same distribution of duplicated hashes as those characteristics. Given the characteristics of a dataset's duplication - the frequency of each duplication count, as seen in Appendix B - it is possible to recreate a representative dataset in order to test new deduplication methods. The original data is represented by unique surrogate hashes of the data.

The tool takes as input the overall rate of object duplication as well as the CDF from section 4.1, expressed as a data characteristics table of the dataset's duplication.

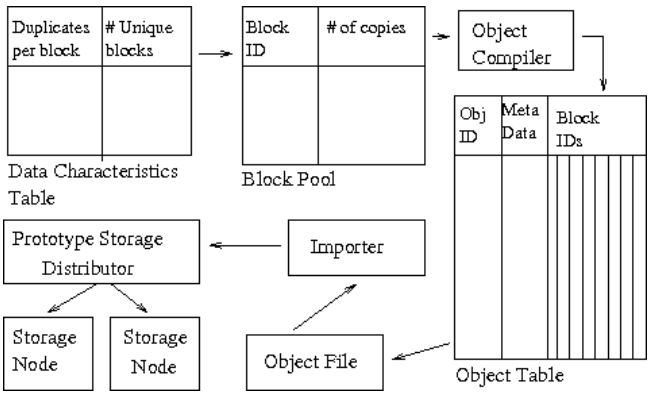

Figure 4-3: The Dataset Duplication Generation Tool. Given data characteristics, a pool of the corresponding number of unique blocks is formed. An object compiler assembles objects of the appropriate object size and object-level duplication from this pool and builds a table of objects. As the table is built it is written out to a file. The file is then imported to a prototype object storage system.

This table contains, for each duplicate count, the number of unique blocks which each has that duplicate count. For example, a duplicate count of 5 with a unique blocks count of 100 would mean that unique blocks with four duplicates each make up 500 of the blocks in the dataset. Using the data characteristics table, the tool then populates a block pool of all unique blocks to distribute, each represented by a unique fake hash which is its ID. Along with the hash for each unique block, the number of copies already distributed and to be distributed is tracked in the pool.

To create object representations, an object compiler takes some specified number of blocks and puts their IDs together to form an object. The object is placed at a random position in the portion of the object table currently buffered in memory. The number of blocks is specified to create the desired object size. If more object duplication is needed to meet the specified full object duplication rate and the object will itself be duplicated some n times, only blocks with n copies remaining to be distributed are selected so that several clones of this object can be placed. When the current object table buffer is full, these object records are flushed to the object table, a representation file on disk using the dataset representation in the section above; the buffer then is filled again. A larger buffer improves the distribution of duplicated objects across the representation file, so buffers one-tenth the size of the full dataset

were used.

This tool could be used on any dataset's characteristics to produce a representation suitable for research purposes beyond this thesis experiment's method of locating and relocating duplicates across nodes in a distributed storage system. It could be useful for data duplication research which would like to use, without need for byte-for-byte access, realistic workloads to determine if some new method may work well with for space savings on that particular workload. The input format used by this tool is fairly simple to create from an existing dataset. In fact, the fields used by the tool may already exist for datasets which have been analyzed for duplication properties. While the generation tool is fairly simple in concept, the implementation becomes quite involved when trying to build a version which can scale to large datasets, in this case over 1TB in size. Any researcher who wants the generation it provides may, rather than build their own, benefit from this tool's use.

4.3 Storage System Setup

This section describes the storage system used for the thesis experiments. This includes a description of the individual nodes and their assembly as well as the process of loading the datasets from the canonical representation into the storage system.

4.3.1 Structure

To test my thesis, I built a prototype simulating a distributed storage system. To build the system, I augmented a storage emulator developed originally by NetApp for their internal use of modeling a distributed storage system with individual nodes. These nodes have the facilities to allocate space and to store objects which they are given. I created my system in C++ as a user-level process. The process reads a configuration file specifying the number of nodes to simulate and the distribution mechanism for objects across the nodes. Based on this configuration, the system then creates the specified number of instances of the storage nodes. This is all performed from the same user-level process, so all communication between the overall system

and each of the nodes is conducted locally by function calls. No RPCs are used. The system-wide deduplication is managed by a global daemon running as a thread at the overall system level.

While the storage nodes were capable of basic object storage, I modified the code in several ways to make deduplication and this experiment practical. First, I added object and block hash fields to the metadata for each object stored in a node. Second, I created a new storage function, which creates only the metadata record for a new object, including hash, but does not use disk space to store the actual data represented. Third, I added functions to the nodes to locate duplicate object internally, as described in the deduplication process section below, in such a way that the full system could have each node report on its internal duplication.

4.3.2 Loading

To load the dataset representation as described in section 4.2.1, the storage system is instantiated with the number of storage nodes as specified in the configuration file. Each line of the dataset representation file is read in and parsed into component attributes, and a request to store an object's metadata is made to a node. In this experiment, the distribution is randomized, as this seemed to be the method which would produce the poorest in-node duplication characteristics for a generic dataset. Because only the metadata was stored and no actual data blocks are written, it was possible to keep many copies of the dataset spread across different numbers of nodes for rapidly running series of experiments.

4.4 Analysis Process

I compared levels of deduplication across each dataset while varying the number of nodes in the system and employing different cross-node deduplication methods. The deduplication process within each node reports on the number of duplicates found, which is aggregated by the global deduplication daemon. This information is collected before and after each experiment to determine the effectiveness of the experiment.

Duplication is reported as the percentage of storage space saved. That is, the number of duplicate objects or blocks which are duplicates of an original relative to all objects or blocks in the original dataset. For example, if a dataset contains only two objects and each is exactly the same, this will count as a single duplicate and be reported as 50% duplication.

Chapter 5

Results

This chapter describes the effects on deduplication from distributing a dataset across varying numbers of nodes and the results of using the cross-node deduplication methods developed in this thesis. In both cases, the level of deduplication is evaluated as a fraction of the total duplicates in the dataset, found by comparing the hashes of all objects present in the dataset. I also analyze the performance impact cross-node deduplication has on the prototype system. I collected data on the levels of deduplication across the nodes of the prototype distributed storage system while performing the experiments already outlined in this thesis. These results were obtained to determine if the reference count hinting system is capable of restoring the duplication space savings lost by distributing a dataset across many nodes.

5.1 Distributed Duplication Characteristics

The first results were collected without using cross-node hinting, locating duplicates only within each node based on object hash. This was done in order to determine the level of space savings lost through distribution, without using my reference count hinting scheme. These results were compared to the savings found by comparing all object hashes across the entire dataset, as in a centralized storage system. In a distributed storage system, the object placement policies which cause duplicates to not be placed within the same node, eliminating potential space savings. I measured

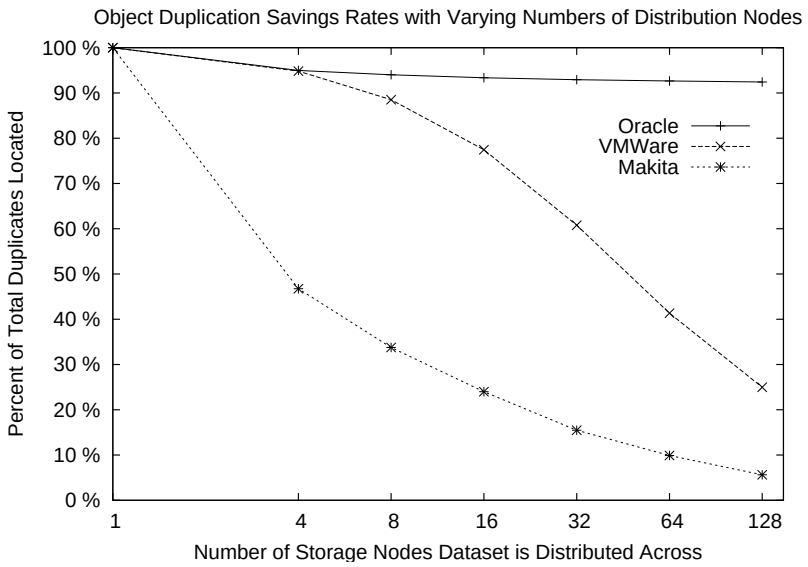

Figure 5-1: Changes in the levels of object duplication space savings as the number of nodes in the storage system increases, compared to the potential savings in a centralized storage system. Deduplication is performed only between objects within each node, without cross-node deduplication.

the effect on deduplication from distributing a dataset across a distributed storage system. The potential savings of a cross-node hinting system is the amount of deduplication space savings that is lost when that dataset is split across nodes. For each of the three datasets whose characteristics are discussed in Chapter 4, a new copy of the distributed storage system was instantiated with 4,8,16,32,64, and 128 storage nodes. The storage system was then populated with the dataset, with random distribution of objects across nodes. Each node performed deduplication internally on the object level as well as on the block level in section 5.3.

As seen in Figure 5-1, the Oracle dataset loses very little space savings as it is distributed across more nodes. This makes sense given that, as seen in Figure 4-2, most duplication in the Oracle dataset comes from a few objects with many duplicates. Datasets with mostly medium and low-count duplication take a very small hit to space savings. Makita, with mostly lower-count duplication has its space savings cut to 1/5 of the original level by distributing across 128 nodes. Similarly, the VMWare dataset, with low and medium count duplicates dropped to just under 1/4 of the original level

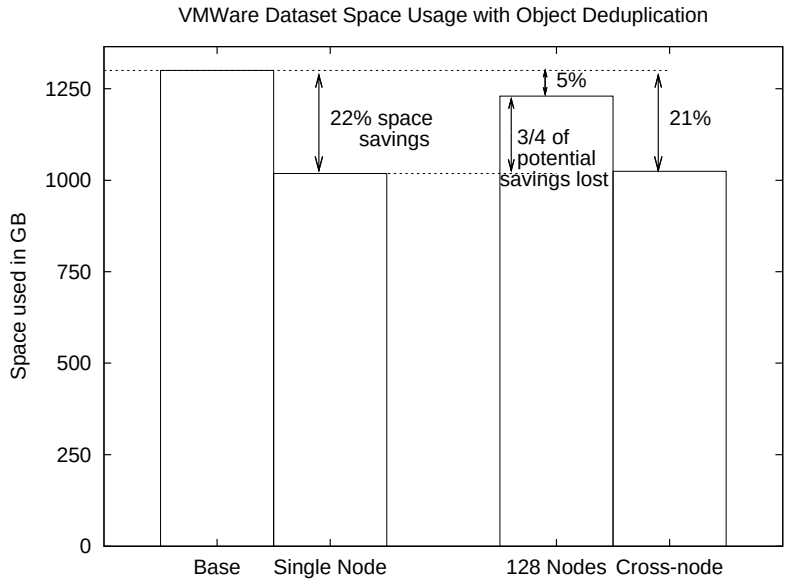

Figure 5-2: Changes in the space used to store the VMWare dataset without deduplication, deduplication with the entire dataset on a single node, deduplication within each node when distributed across 128 nodes, and deduplication within each node when distributed across 128 nodes after cross-node moves due to object-level hinting.

of space savings. With the exception of the Oracle dataset, at least 1/2 of space savings is lost with distribution across more than 32 nodes, indicating that file and virtual machine storage has high potential for space savings with a cross-node hinting system.

5.2 Object Hinting

I performed experiments using the proposed cross-node hinting method on datasets distributed across 128 nodes to determine what fraction of the potential duplicates the hinting scheme locates. I selected this number because it was on the tail end of distribution losses and makes each node store 10GB of data, the target size for the experimental system under development at NetApp. In the accompanying graphs, the "Base" bar indicates the size of the dataset without any deduplication while the "Single Node" bar indicates the size with all potential duplicates found, as in a centralized storage system. As seen in the results for the VMWare dataset in Figure

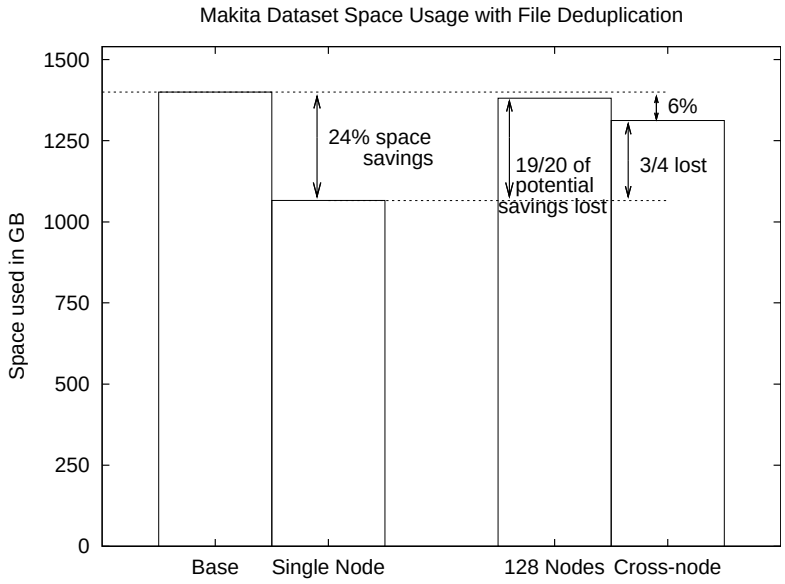

Figure 5-3: Changes in the space used to store the Makita dataset without deduplication, deduplication with the entire dataset on a single node, deduplication within each node when distributed across 128 nodes, and deduplication within each node when distributed across 128 nodes after cross-node moves due to file-level hinting.

5-2, the 22% difference between the "Base" and "Single Node" bars is the maximum savings inherent in the data through object deduplication. This is the maximum level the hinting method would like to attain. The 5% savings on the "128 nodes" bar is the space savings using only in-node object deduplication as in Section 5.1, after distributing the dataset across 128 nodes, representing a loss of 3/4 of the potential savings.

After identifying duplicates within each node and locating duplicates of these across other nodes, duplicates of each object were moved to the same single node and the new in-node duplication level examined again. This is the 21% shown on the "Cross-node" bar. This represents only a loss of 2% compared to the "Single node" savings, which is expected given that the greatest amount of duplication in this dataset comes from objects duplicated around 100 times. Refer back to Figure 4-2 for more details. The time and space costs associated with this space savings will be discussed later, in Section 5.4.

The Makita dataset, at 24% space savings, has similar potential object duplica-

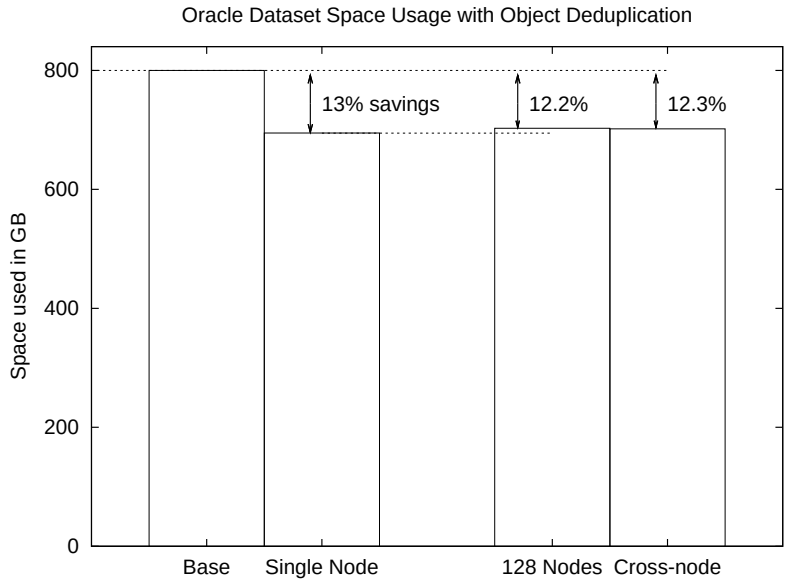

Figure 5-4: Changes in the space used to store the Oracle dataset without deduplication, deduplication with the entire dataset on a single node, deduplication within each node when distributed across 128 nodes, and deduplication within each node when distributed across 128 nodes after cross-node moves due to object-level hinting.

tion to VMWare, but loses much more savings, down to 1%, when distributed with deduplication performed only within each node based on object hashes. This is due to the lower portion of objects with high duplicate counts. Similarly, when using hinting to move duplicated objects across nodes, duplication space savings grows only to 6%. While this is only 1/4 of the potential savings, it represents an increase to five times greater space savings than without cross-node hinting. The reason why the hinting technique fails to find most of the deduplication opportunities is that the Makita dataset has a majority, about 3/4, of savings in objects duplicated less than 20 times, as outlined in section 4.1. So, thse objects were less likely to be found duplicated within any given node and thus there was no attempt to deduplicate them across nodes. With non-randomized distribution, locating duplicates within nodes may be more likely, resulting in greater space savings, but given the large portion of low-count duplication, much more savings should not be expected using this method on the makita dataset.

The Oracle dataset has only a 1% loss of space savings when moving from a single

node to 128 nodes, leaving little room for improvements through cross-node hinting. This lack of loss comes from the duplication characteristics of the Oracle dataset, with most duplication savings found in objects with at least 20 million duplicates. This leaves the deduplication within nodes alone as sufficient, given that 128 unique copies of an object are negligible by comparison. In my experiment, only a 0.1% gain was seen by using the cross-node hinting mechanism, because the predominance of high-count duplicates left little room for improvement at all.

Overall, whole object deduplication-based hinting and cross-node object movement achieves much better space savings than the space savings in a distributed system without these methods, where deduplication is performed only between objects within the same node. It is also possible that after using these methods, a storage node could perform another form of deduplication internally, such as on a block level, potentially achieving even greater space savings. In Section 5.4 we describe the run-time overheads associated with object movement.

5.3 Block Hinting

Deduplication with hinting based on individual 4KB blocks has the potential to find duplicates missed by the full object method. This is because there are more blocks than full objects, so there is a greater probability of finding at least one block of an object larger than 4KB duplicated within a single node than the whole object. I constructed an experiment similar to the extent-based deduplication hinting above, but using hinting based on duplication of individual 4KB blocks within each node instead of full objects. This change increases the number of hashes being checked for duplicates in each node by two orders of magnitude. Due to the prototype implementation running on only a single computer, there was insufficient memory to handle this duplicate identification for each node as well as the overall deduplication daemon. This supports the motivation of this thesis, that full tracking of every block in a distributed storage system by some central node is too resource intensive to be practical, which is what the prototype was really attempting to do.

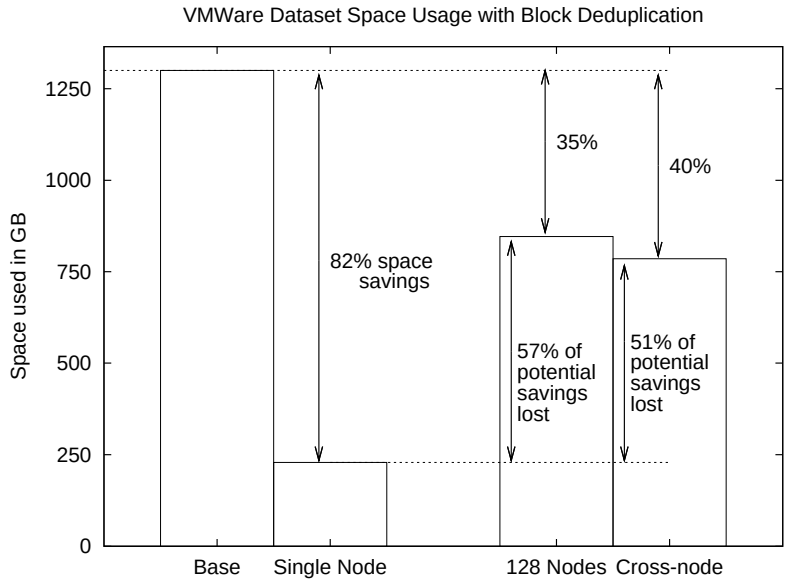

Figure 5-5: Changes in the space used to store the VMWare dataset without deduplication, block deduplication with the entire dataset on a single node, block deduplication within each node when distributed across 128 nodes, and block deduplication within each node when distributed across 128 nodes after cross-node moves due to object-level hinting.

Partial results were obtained for block-level deduplication with the VMWare dataset, as seen in Figure 5-5. The first three bars are as in the graphs above, with the 35% savings with 128 node distribution representing a 57% loss of savings over the 82% ideal block deduplication savings. While it was not possible to run the experiment using block hinting, I was able to obtain results on the level of block deduplication after the extent-based hinting from the section above, with 40% savings representing only 51% lost. This small gain in space savings is due to the mismatch between hinting and moving at the object level but deduplicating at the block level. While a duplicated object was moved between nodes its component blocks may have been duplicates of other blocks in its original node, possibly causing a loss of space savings rather than a gain. This is not the case for all objects, so there is still some gain, but it is not as great a portion of potential savings as that gained when both hinting and deduplicating with objects.

Node Hashes Before	10,000
All Interest Objects	4,500
Objects Moved	284,000
Objects Saved After	289,000
Node Hashes After	7,750

Table 5.1: Overhead statistics for VMWare dataset distributed across 128 nodes.

5.4 Overhead

Cross-node hinting, as with any form of deduplication, will require some overhead both in data structures and the time taken to run any deduplication algorithms, but this per-node hinting method attempts to limit the amount done by any given node in the system, including the central deduplication daemon. In order to identify duplicates within each node, step 2 in Figure 3-1, the set of all object hashes is stored in memory as an ordered set, along with a mapping from each object hash to its ID within the storage node for faster access. In the VMWare dataset, as seen in Figure 5.1, this is a set of about 10,000 hashes per node. Set insertions take $\log n$ time, so given some o objects in the system, inserting them into a set can be done in $O(o \log o)$ time.

In step 3, the central deduplication daemon collects the duplicates identified by each node, removing any duplicates already identified by another node, and storing only the unique interesting object hashes, u, across the entire system, only $4,500$ in the VMWare dataset. Each node is then asked in step 4 to locate any matches to the duplicates identified in other nodes, which can be done by comparing those u object hashes to the set of hashes already stored by each node. This can be performed in $O(u \log u)$ time.

After collecting matches between nodes in step 5, the global deduplication daemon asks all but one node with each duplicate to move that duplicate to that one node in step 6. My naive implementation simply had the first node to identify each duplicate internally take the duplicates from the other nodes, such that a node with 8 copies of an object internally might be asked to move them to another node with only 2 copies rather than the other way around simply because one was identified after the other.

Node Hashes Before	89,400
All Interest Objects	61,600
Objects Moved	2,000,500
Objects Moved	86.5GB
Node Hashes After	80,400

Table 5.2: Overhead statistics for Makita dataset distributed across 128 nodes.

In the VMWare dataset, this resulted in 284,000 objects moved between nodes, which is approximately 284GB of data, representing the majority of the 289,000 duplicate objects identified after this process completes. After this process is complete, each node continues to store a set of all hashes internally, but due to the aggregation this averages only 8,400 per node, consuming very little memory.

The prototype implementation ran only once across the dataset in a single-threaded manner such that each node was asked to identify duplicates one at a time. In a full implementation, this would be done in parallel by all nodes at once, adding no computation or storage factor per node as more nodes are added to the system. Further, with a system running continuously and a periodic deduplication cycle, identification of duplicates need only be performed on or with objects added to each node since the last cycle, cutting the time required.

The overhead involved in deduplication for the Makita and Oracle datasets is found in Figures 5.2 and 5.3. While the Makita data is similar to that of the VMWare dataset, for Oracle nearly the entire set of duplicates was moved between nodes, for very little gain, due to the Oracle dataset having so much duplication from a few objects duplicated many times. This could be avoided with a more intelligent method for determining moves after duplicates across nodes are located, by keeping track of the number of duplicates already present in each node and setting a maximum level at which there would be a move.

Node Hashes Before	5,600
All Interest Objects	71
Objects Moved	99,700
Node Hashes After	5,600

Table 5.3: Overhead statistics for Oracle dataset distributed across 128 nodes.

Chapter 6

Conclusion

As storage systems grow, it is necessary to distribute data over many storage nodes. While some systems base their object distribution on content, this is a secondary consideration in other systems, so potential space savings is lost as duplicates are placed on different nodes. This thesis examined a method with little overhead for regaining space savings by locating duplicated data across nodes.

Identifying and moving duplicate full objects between nodes, as demonstrated with the VMWare dataset, can achieve nearly the same level of object duplication space savings as is possible in a centralized system. As duplicate objects are identified, moving all copies of that object to a single node requires little decision-making. By contrast, if duplicate blocks are identified, even given sufficient memory to track all of those blocks, deciding which object moves would result in the greatest space savings is an NP-hard problem. Meanwhile, if object-based deduplication is already happening within each node, there is little extra memory overhead involved in the cross-node duplicate identification.

Using objects for hinting between nodes does not restrict a node's internal deduplication. A node can use block-level deduplication internally and make use of space savings both from duplicate blocks in the duplicate extents as well as anyother block duplication present. Using block or object-based deduplication internally, object-based cross-node hinting can provide improved space savings with little overhead in a distributed storage system.

Chapter 7

Related Work

I consulted many papers in the development of this thesis. The theoretical distributed system for which the hinting system was developed is largely based on Ceph and Hydrastore, which are similar distributed storage systems with a focus on deduplication. The Data Domain paper referenced made use of caching in a way which indicated that hinting could be successful.

7.1 Ceph

Ceph [7] is an object storage file system which decouples the storage of objects from the metadata about those objects. When a file is requested, the Ceph client contacts a metadata server to open the file. The metadata server is able to respond quickly, and the client is not required to contact the object server actually holding the data; rather, the client is given the file inode which the client can use to determine which placement group the file is in. Mapping of files to placement groups is done with a consistent hashing algorithm operating on the file's inode, which makes file access simple, though it does reduce the potential node locality of multiple related files. From placement groups, of which there are around 100, clients are kept aware of which object servers are hosting which placement groups. This extra layer of abstraction is useful for allowing dynamic allocation of resources and adding new servers.

This is a simple distributed object storage system which makes deduplication

possible across nodes by managing the distribution of objects across nodes based on content. While each node is able to do the deduplication internally, seeming to solve the problem this paper addresses, this is only possible because the clients perform an expensive hash calculation. This thesis instead assumes the system must be addressable by existing filesystem protocols, making the former approach impractical. Nevertheless, Ceph stands as a good reference on how a distributed filesystem with deduplication as a high priority might work.

7.2 Data Domain

The Data Domain file system [9] is a deduplicating file system designed for streaming data such as backups. As data comes in, it is broken into variable-length segments by content. A summary vector, which uses a Bloom filter of SHA-1 hashes of the segments, is queried to determine if the segment should be stored or if there may be a duplicate of this vector already stored. If the segment is to be stored, it is placed in the next available container. Containers are written to serially until filled and cannot have data removed, such that if data is read back in the same order it was written caching of containers will result in fast reads. If the summary vector indicates that a duplicate segment has already been stored, each container's metadata is checked for a matching segment hash and when found the container for the new segment references the container for the existing one. Due to the stream locality within containers, the next segment to be written is also likely to be deduplicated against a segment in the same container, so caching of container metadata speeds up the deduplication process while writing.

While the streaming locality properties do not apply to a general object store, they show that if one segment in a file has deduplication against another file, subsequent segments may as well. This could be useful in reducing the work of finding potential deduplication between nodes in a distributed system under a filesystem workload similar to the file-based hinting described in Chapter 3 of this thesis. The use of a Bloom filter to make central tracking of possible deduplication also seems like a useful

structure. Taking file sequentiality more generally as metadata about an object, this paper shows that given some sampling, if two objects are duplicates of each other and have some metadata which matches, then other objects which match on that bit of metadata are more likely to be duplicates as well, which is the basis of this thesis.

7.3 HYDRAstor

HYDRAstor [3] is a deduplicating secondary storage system designed for fast writing of long streams of data across a grid of storage nodes. As data comes in, it is divided into variable-sized blocks based on content, and an SHA-1 hash is taken. Based on the prefix of the hash, the supernode for that block is selected and checked to see if a duplicate exists, in which case a pointer is inserted to the existing block. Otherwise, the block is distributed in fragments across the peer nodes of that supernode. These fragments are created using erasure codes with degrees of redundancy selected on a per-block level.

Like Ceph, HYDRAstor provides deduplication as a primary goal of the storage system, similarly distributing data across nodes based on content. While this system does not require client hash calculation and awareness, the calculation is still placed on the highest level, producing a potential bottleneck which would not be tolerated in a performance-focused system. The fragment distribution could be used to improve deduplication across nodes in a hinting system because each fragment is small, so there are more chances for duplicates being located within a node, with a probability greater than that of a full object's being duplicated within a node, as tested in this thesis. That information could then be used to deduplicate other fragments of the same object between other nodes. This, however, would be dependent on the implementation of storage at each node and is not broadly applicable.

Appendix A

Dataset Representation

The text below is a sample of four objects in the dataset representation format. Each line represents a new object. The fields are: name, hash, size, permissions, block hash, block hash,...,block hash


```
/html//home/index.html,1222dbbb99b88c5abdb28ecda3bd0a96,66538,644,469
ea2c2579b81326a6630596efec79b,9e4e51024b0af29493b24ee18b6323c8,7c0decc0657ca1e38
b5ed41f12ab505c,17576e3dc8cd338df8889aca57b8cdd4,66b362ff84762bb3541c5df3e963879
9,7cf5c500f2d305be8050960e81b7e638,e60bed5770b02b18ee3ebddc4ae91d6f,835a702a7ee1
0ea04fd1fdf7225a162,d3d1f9b41607a94bc0b6194ca087d43,8319b6fa4a9ae801810129a9c67c
1cbd,22b2fc4b324da7a13d5192ea7f4b2142,5a67c9e474f9f2a991765e712910c156,5a3966db1
a4b5ab9363fe963c9afb45d,346e1545ce5b67769f5aafd415cab30,a51202ccf2358574d3050863
8c3d2b0,602b3372d4872e71f037aec2d26c81ff,83c9c320a626ec8f6296da4bd863ba11


/html//home/index.html.old,e90ef6b3666d70344d7a65eeb7419ba4,66531,644
,b0b7220453edd7bd160ffd9c481b3556,a2ede70d22ac564faa9f6a8756808688,bdbfdceeb29eb
12ede12badb0a6cee8,d9212c4c4a382507e2597b572b621a18,26754bd2279d55d7b7db50162a68
5108,27c87f8cc21a25281f716272d846703,dddb305ff821e0f048ca863e7cf3c33c,9178f6bda3
8b6804c06e155bc6c7d242,a1d8c2b26bf25bf5a2b72fc532f0eaf0,670cbe97eba13bbaf0265c21
cd0d50e0,77d9f4c54045d28787c6d1a830e2d2,d548aed420cf0b0ec32322e8d06df821,edcea98
33fe93ec1f061c6244c68067f,45018a681e7541a48909b805dc115771,3737ce92ac425a17c684b
7ead953aa08,5ca5ed1a3500fe7597ad0e74ca2f62e,eab0853eb89956a39e5ecdc0ec8e337d


/html//home/index_alpha.html,392bb0e6883b5d191a085f0775c0f17f,75892,6
44,9d8224e2efd0f7687c70325c3e9bedbd,d1a1ba896e194f550ccee118eda6c0,ef35da35ee973
720ba5aff8af5fef1e4,10884be51b6886ad352fe5fccd00ed87,a79bd46f6cc8cbd3a91e1f14ed2
b18ca,4562d1df835fa929f44068ad6dc86b98,c178ba392f99fb6797932297f0988583,831f975f
38b15814758803f153a5f30,2e54744c2147189299188bd7b1fa5a99,edfeadb8ad97232442f59a4
756612cd8,ca00fcb5b7f1ec6c2f5fa9eca9174be1,99e58129e2e668e571481ea9ef8e7aca,c0b8
615b663a4807d3fdce2f79000305,61fc517b834ef130f1186cb476406c14,435663ca63794f44a0
78757035782f4d,1fa397d4b7580b2bc21430bdbafbdc56,68028d908d3432bdc4019f82511380d4
,4891a3dc9a14b575808b885b235e4840,ac8dc5f9bf1eddab6df2226da37edfcc


/html//home/.DS_Store,e407d022ebd2484fdd09d819300c90e4,6148,777,82c9b
e8713bbd3edcbb5e3af8efdc4f,52bf97f1529aa9e9b7c7e47dd548cb0
```

Appendix B

Generation Tool Input

This is a sample of the first thirty lines of the input to the Duplication Dataset Generation Tool for the VMWare dataset. The right column is the count of instances for each block, while the left column is the number of unique blocks with that count of instances. This data can be graphed to produce the CDF of space savings found in figure 4-1.

Uniques	Count
45470312	1
6555889	2
2008324	3
1289601	4
819171	5
589635	6
421182	7
333212	8
258537	9
174660	10
241312	11
152619	12
154370	13
142634	14
132451	15
134092	16
98119	17
72556	18
62479	19
55415	20
37812	21
40887	22
46013	23
51424	24
38973	25
53168	26
39507	27
39788	28
34637	29
26617	30

Bibliography

[1] WJ Bolosky, S Corbin, D Goebel, and JR Douceur. Single instance storage in Windows (R) 2000. In *USENIX ASSOCIATION PROCEEDINGS OF THE 4TH UNSENIX WINDOWS SYSTEMS SYMPOSIUM*, pages 13–24, SUITE 215, 2560 NINTH ST, BERKELEY, CA 94710 USA, 2000. Usenix Assoc, USENIX ASSOC. 4th Usenix Windows Systems Symposium, SEATTLE, WA, AUG 03-04, 2000.

[2] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, June 2009.

[3] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: a scalable secondary storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[4] Bill May. Netapp deduplication for fas. *NetApp Technical Reports*, (TR-3505), April 2008.

[5] C Policroniades and I Pratt. Alternatives for detecting redundancy in storage systems data. In *USENIX ASSOCIATION PROCEEDINGS OF THE GENERAL TRACK 2004 USENIX ANNUAL TECHNICAL CONFERENCE*, pages 73–86, SUITE 215, 2560 NINTH ST, BERKELEY, CA 94710 USA, 2004. USENIX Assoc, USENIX ASSOC. 2004 USENIX Annual Technology Conference, Boston, MA, JUN 27-JUL 02, 2004.

[6] S Quinlan and S Dorward. Venti: a new approach to archival storage. In *USENIX ASSOCIATION PROCEEDINGS OF THE FAST'02 CONFERENCE ON FILE AND STORAGE TECHNOLOGIES*, pages 89–101, SUITE 215, 2560 NINTH ST, BERKELEY, CA 94710 USA, 2002. USENIX Assoc, USENIX ASSOC. Conference on File and Storge Technologies (FAST 02), MONTEREY, CA, JAN 28-30, 2002.

[7] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Usenix Association 7th Usenix Symposium on Operating Systems Design and Implementation*, pages 307–320, SUITE 215, 2560 NINTH ST, BERKELEY, CA 94710 USA,

2006. USENIX Assoc, USENIX ASSOC. 7th USENIX Symposium on Operating Systems Design and Implementation, Seattle, WA, NOV 06-08, 2006.

[8] Eric W. Weisstein. Birthday problem. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BirthdayProblem.html, December 2009.

[9] 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 269–282, SUITE 215, 2560 NINTH ST, BERKELEY, CA 94710 USA, 2008. USENIX; NetApp; Google; Microsoft Res; pdsi; Seagate; SNIA; hp invent; IBM; Natl Sci Fdn; Sun Microsyst; Yahoo, USENIX ASSOC. 6th USENIX Conference on File and Storage Technologies, San Jose, CA, FEB 26-29, 2008.