Improving Block Sharing in the Write Anywhere File Layout File System

by

Travis R. Grusecki
S.B. Computer Science and Engineering & Mathematics, MIT 2011

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 2012

Author: _____
Department of Electrical Engineering and Computer Science
May 21, 2012

Certified By: _____
M. Frans Kaashoek, Professor
Thesis Supervisor
May 21, 2012

Certified By: _____
Paul Miller, Software Engineering Manager (NetApp, Inc.)
Thesis Co-Supervisor
May 21, 2012

Accepted By: _____
Professor Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Improving Block Sharing in the Write Anywhere File Layout File System

by

Travis R. Grusecki
S.B. Computer Science and Engineering & Mathematics, MIT 2011

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

Abstract

It is often useful in modern file systems for several files to share one or more data blocks. Block sharing is used to improve storage utilization by storing only one copy of a block shared by multiple files or volumes. This thesis proposes an approach, called Space Maker, which uses garbage collection techniques to simplify the up-front cost of file system operations, moving some of the more difficult block tracking work, such as the work required to clean-up after a file delete, to a back-end garbage collector. Space Maker was developed on top of the WAFL file system used in NetApp hardware. The Space Maker is shown to have fast scan performance, while decreasing the front-end time to delete files. Other operations, like file creates and writes have similar performance to a baseline system. Under Space Maker, block sharing is simplified, making a possible for new file system features that rely on sharing to be implemented more quickly with good performance.

Acknowledgements

My parents, Rich and Lori Grusecki, who couldn't have been more supportive of my studies and work at MIT and NetApp.

Frans Kaashoek, my thesis supervisor and TA supervisor for 6.033. Besides being an integral part of helping me complete my thesis, he is also the one of the people who first stimulated my interest in computer systems.

Paul Miller, Sean Smith, and the rest of the WAFL Space team at NetApp for helping me to develop my thesis and providing me the necessary support to see my work through to completion.

Vsevolod Ivanov and Tanya Kortz, my two closest friends at MIT. Between the late-night study parties and random hacking, they are the people most responsible for me having a productive yet enjoyable time at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In modern file systems, it is common for multiple files to share several data blocks in common, a property that is more generally referred to as *block sharing*. Properly leveraging block sharing improves storage utilization, allowing users to store more data in less space. Block sharing in high performance file systems has always been a difficult problem because it is challenging to maintain accurate block pointers that span fragmented parts of the file system. Often different features need to share blocks for different reasons, usually with different performance and usability constraints. The goal of this thesis is to explore a new approach for improving block sharing in a copy-on-write file system.

For the purpose of this thesis, the work being performed uses a commercial system produced by NetApp, Inc., a leading manufacturer of data storage appliances. This thesis research is being conducted through the MIT VI-A program, which allows a thesis to be completed in conjunction with a partner company, which is NetApp in this case. The research focuses on improving certain aspects of the file system used in storage appliances at NetApp.

The Write Anywhere File Layout (WAFL) file system is the file system used by NetApp. WAFL has the unique property that blocks on disk are never overwritten; rather, data is always written to a new distinct location. The old data can safely be removed once the write completes or it can be protected in a snapshot for later recovery. While this architecture leads to several obvious performance and reliability improvements, it also creates fragmentation issues.

Modern file systems like WAFL often provide features that allow a customer to save disk space. One such feature, *data de-duplication*, scans the file system looking for files with

identical data blocks. In the case where common blocks are found, the file system transparently removes all but one copy of the common block, which is then marked as copy-on-write. Another feature, *volume cloning*, can create an entire copy of a volume without consuming additional space. This is accomplished by marking all blocks in the volume as copy-on-write, enabling the cloned volume to only use the space necessary to store differences form the origin volume.

As new features come to market like data de-duplication and volume cloning, it is increasingly necessary for multiple files to share the same data blocks to get high storage utilization. Furthermore, it is desirable to be able to share all file system blocks, even those in metadata trees, rather than just data blocks. This thesis seeks to address this challenge by identifying an approach for improving block sharing performance while not significantly impacting other parts of the file system.

While several techniques were considered, the most promising is an approach which removes the requirement to track block sharing during front-end operations like deletes and clones, in favor of using a background garbage collector that scavenges the file system for blocks that are no longer in use. This system inherits some of the garbage collection ideas of LFS [5] while preserving the general architecture of WAFL. This approach, called Space Maker, is shown to have high delete performance while not affecting standard file write performance. Additionally, the background garbage collector is shown to have good performance under a variety of workloads. In the chapters that follow, this approach will be described in detail along with a description of a prototype which is shown to meet these performance objectives.

Chapter 2

Background

The difficulty of dealing with large amounts of block sharing is a widespread issue in modern file system. Understanding this issue in the context of NetApp filers requires a greater understanding of how NetApp's file system uses and manages data.

2.1 WAFL Layout

WAFL shares many design elements with standard UNIX file systems. Like other mainstream UNIX file systems, the inode represents a file. Inodes point to indirect block trees and subsequently data blocks. In the case of very small files, all of the necessary data blocks are directly referenced from the inode.

WAFL's largest file system unit is the *Aggregate*. An Aggregate is a RAID group of disks combined into a single common pool of storage. Most NetApp filers use either RAID-4 or RAID-DP, a proprietary version of RAID-4 with two parity disks [1]. An Aggregate can contain one or more *Volume* objects. User data can only be stored inside a volume. **Figure 2-1** depicts a sample Aggregate layout with several volumes.

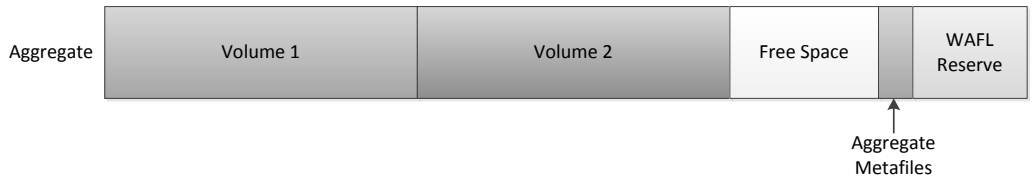


Figure 2-1: Diagram of sample WAFL Aggregate including two Volumes

WAFL reserves 10% of each Aggregate as a "WAFL reserve". As volumes are created in an Aggregate, a proportional amount (10% of volume size) of WAFL reserve is also allocated for the volume.

WAFL uses several internal structures to maintain its internal metadata. At the top is a `Vol_info` block, which stores information about the volume and a copy of the RAID label information from the Aggregate [1]. The root of the file system is the `fsinfo` block, which stores file system information, user-defined parameters, and storage counters [4]. These blocks comprise the volume header information. Like most UNIX file systems, a block map maintains a consistent state of which blocks are currently in use. This "*active map*" tracks blocks in use by the active file system while the *summary map* tracks which blocks are in use by file system snapshots [4]. If a block is marked as not in-use by both the active and summary maps, it is made available to the write allocator.

Also, like most UNIX file systems, an indirect block tree provides a hierarchy to organize file system data blocks. **Figure 2-2** provides a depiction of how the WAFL file system is constructed for a single volume.

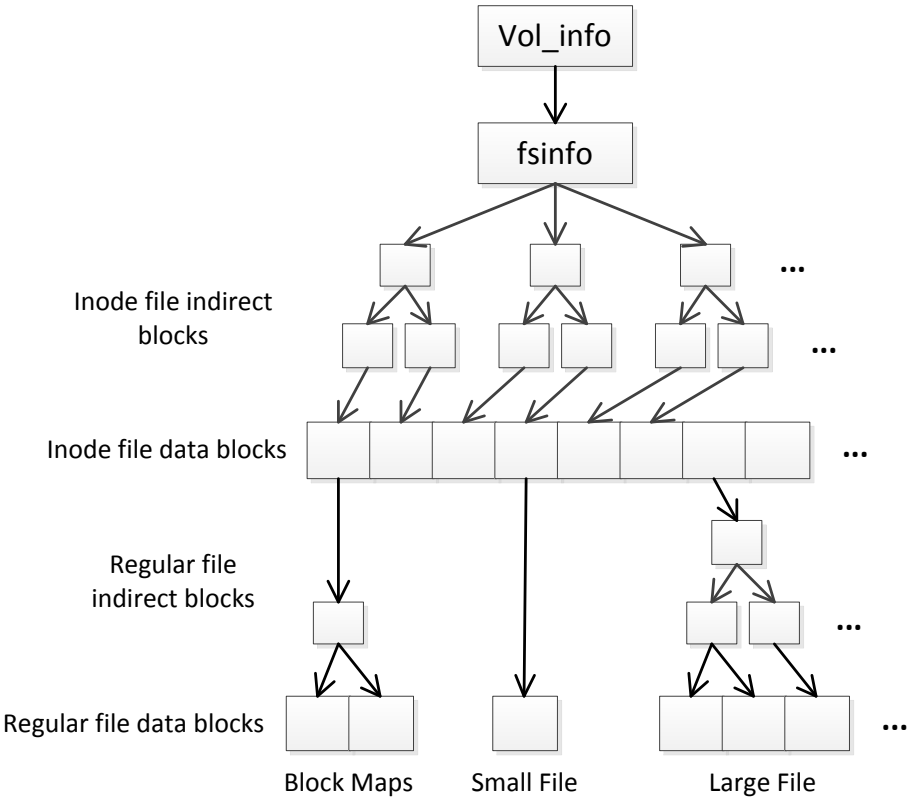


Figure 2-2: Diagram of WAFL file system layout

WAFL is unique from other file systems in the manner in which it writes data to disk. All NetApp filers have non-volatile RAM (NVRAM) which buffers writes. WAFL gains a few nice advantages by batching writes in this fashion. First, data is protected from loss due to power, network, and disk failures during writes. Second, all write operations complete faster. Since operations in NVRAM are much quicker than hard disk operations, a write request can succeed from the OS's perspective nearly instantaneously.

As is depicted in **Figure 2-3**, the large size of NVRAM allows the filer to fill a full RAID stripe (across all data disks) in one data write operation. This provides a significant performance advantage because the parity disk is computed once for data simultaneously written to all disks in the stripe. Data is collected until an NVRAM segment is filled in the standard case. While this write and parity computations are occurring, new file operations are batched in the other half of NVRAM. The NVRAM size is sufficiently large to ensure that the filer can keep up with a demanding workload. In some scenarios involving high availability and clustering, the size of NVRAM segments is much smaller.

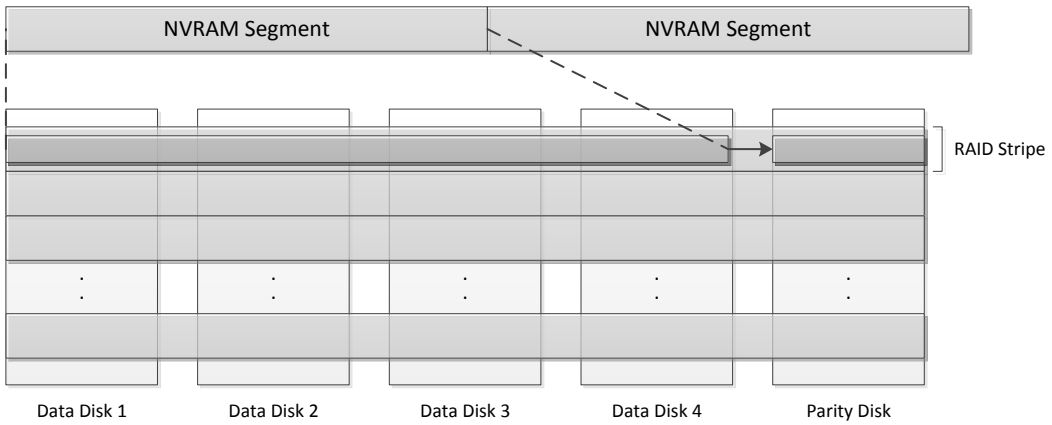


Figure 2-3: Diagram of RAID-4 array with NVRAM

Periodically, the contents of NVRAM are transferred to disk. This process is referred to as a *Consistency Point* [1]. During this process, new blocks are added to the file system tree for new data as well as the associated indirect blocks, but old pointers are maintained for unchanged blocks. This process is depicted in **Figure 2-4**.

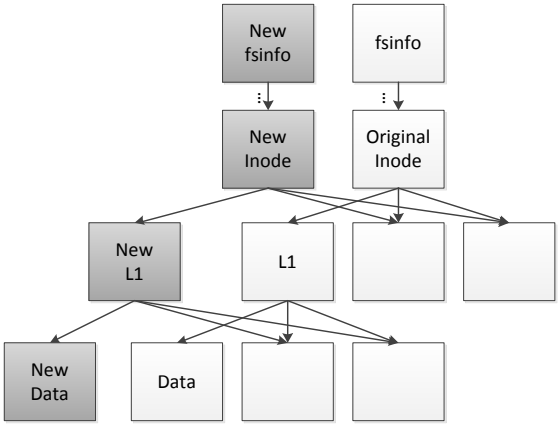


Figure 2-4: Diagram of a Consistency Point: The blocks in gray represent modified data, while the white blocks represent data already present on the disk.

In **Figure 2-4**, a single data block is modified, causing it and its associated file system metadata blocks to be rewritten to disk. The top-level metadata blocks have to be updated as well to maintain volume and space usage counters. It is not necessary, however, to rewrite unchanged data or metadata blocks to disk because block pointers to the old blocks are maintained. The superseded L1 and Data blocks can either be removed after the write completes or they can be protected in a snapshot.

Chapter 3

Block Sharing Challenges

NetApp filers are high performance, feature-rich storage appliances. One of the main selling points of these appliances is their "storage efficiency", or their ability to store more data in less disk space. The main technique for accomplishing this is consolidating identical blocks, using data de-duplication techniques. This process creates block pointers throughout the file system so that identical blocks are only stored once. Other NetApp technologies like volume clones and snapshots also create shared block pointers. In highly redundant systems (like those that host many similar virtual machines), it is possible for some blocks to be shared hundreds or thousands of times.

Currently data de-duplication, volume clones, and snapshots all have separate approaches and data structures for identifying, allocating, and accessing shared blocks. This is undesirable as a development practice, especially as customers demand additional features in the file system. For example, a potential application is file-level snapshots. Currently, snapshots are only taken at the volume level. It is certainly foreseeable that a storage administrator might want to enable snapshots only on important or commonly modified files. Additionally, administrators may desire automatic management capabilities over these new snapshots. Currently, there is no obvious manner in which to implement this feature.

The goal of this thesis is to present an approach to make feature-agnostic block sharing more feasible and scalable without sacrificing overall file system performance. While it is not the goal of this thesis to implement any new features that might leverage a new block sharing approach, it is clear that a simple block sharing scheme, like the one described in this thesis, will make implementing these kinds of features simpler.

Chapter 4

Previous Work

The problem of efficiently sharing blocks among potentially thousands of files has become an area of great research interest. Specifically, the need for large-scale data de-duplication and volume cloning with thousands of volumes has accelerated the need for a better scheme than standard data block level reference counts. The most recent approach was proposed by a team from Harvard and NetApp that leveraged a scheme they referred to as log-structured back references. Taking cues from long-standing research in log-structured file systems, they proposed tracking back references from data blocks to their users using a log [3].

The general approach of log-structured back references is to create two tables to track references between blocks. A `From` table tracks when a sharing reference is created and a `To` table tracks when it is destroyed. When a block is in use, it will have an entry in the `From` table without a corresponding entry in the `To` table. Once a block is freed, it will contain a reference in both tables. A join on these tables is then performed to determine which sharing references are still valid and which can be safely discarded.

The log-structured scheme is a useful start, but it does not meet all of the properties one might want in a new block sharing approach. First, it incurs an overhead for virtually all file operations. While the overhead is relatively small (usually less than 7%), this can add up in systems with many thousands of operations happening per second. Secondly, it shows good performance only when files are located sequentially on disk, since adjacent table entries often correspond to adjacent files. While its performance is acceptable in cases with many contiguous files, this may not be suitable for a file system in which the inherent design creates heavy fragmentation.

Additionally, this approach creates a significant amount of metadata in the form of the `To` and `From` tables. Macko et al. assert that the metadata does not need to be space efficient because storage is inexpensive. This might be true in some systems, but many of NetApp's customers run with nearly full volumes on a regular basis, meaning the large amount of metadata might be prohibitive for some customer scenarios. It is for these reasons that additional work needed to be done to locate a better way of sharing blocks that makes use of the unique structure of modern file systems like WAFL.

Chapter 5

Considered Solutions

Several approaches have been considered to enhance block sharing in the WAFL file system. These approaches represent options with various tradeoffs for solving this problem. The goal of this thesis was to identify and implement one of these approaches after identifying the performance and scalability tradeoffs of each.

5.1 Structured Reference Counting

One approach involves adding reference counts to every block of the file system. By allowing blocks to be shared at all file system levels, this approach makes sharing entire files or chunks of files possible. This would greatly simplify adding features like file snapshots. The approach has the distinct advantage of not requiring a significant amount of engineering since it only requires storing reference counts either in a centralized reference count file or in more localized block-specific metafiles. Implementing this approach would also require modifying the write-allocator to make these updates.

Some work would be required to rewrite other WAFL features to take advantage of this new capability to realize additional block savings. It is not clear what kind of performance this approach will yield. Keeping track of these new reference counts will require significantly more updating when file contents change, which may prove expensive under certain workloads. **Figure 5-1** depicts how this approach would impact the file system.

While reference counting has the advantage of simpler implementation, it has the obvious shortcoming of requiring significantly more metadata updates for each write. Under this scheme, it would be necessary to load either a large file-system wide reference count metafile or one of many localized reference counting metafiles during every write, even those for new files, which would significantly slow write operations.

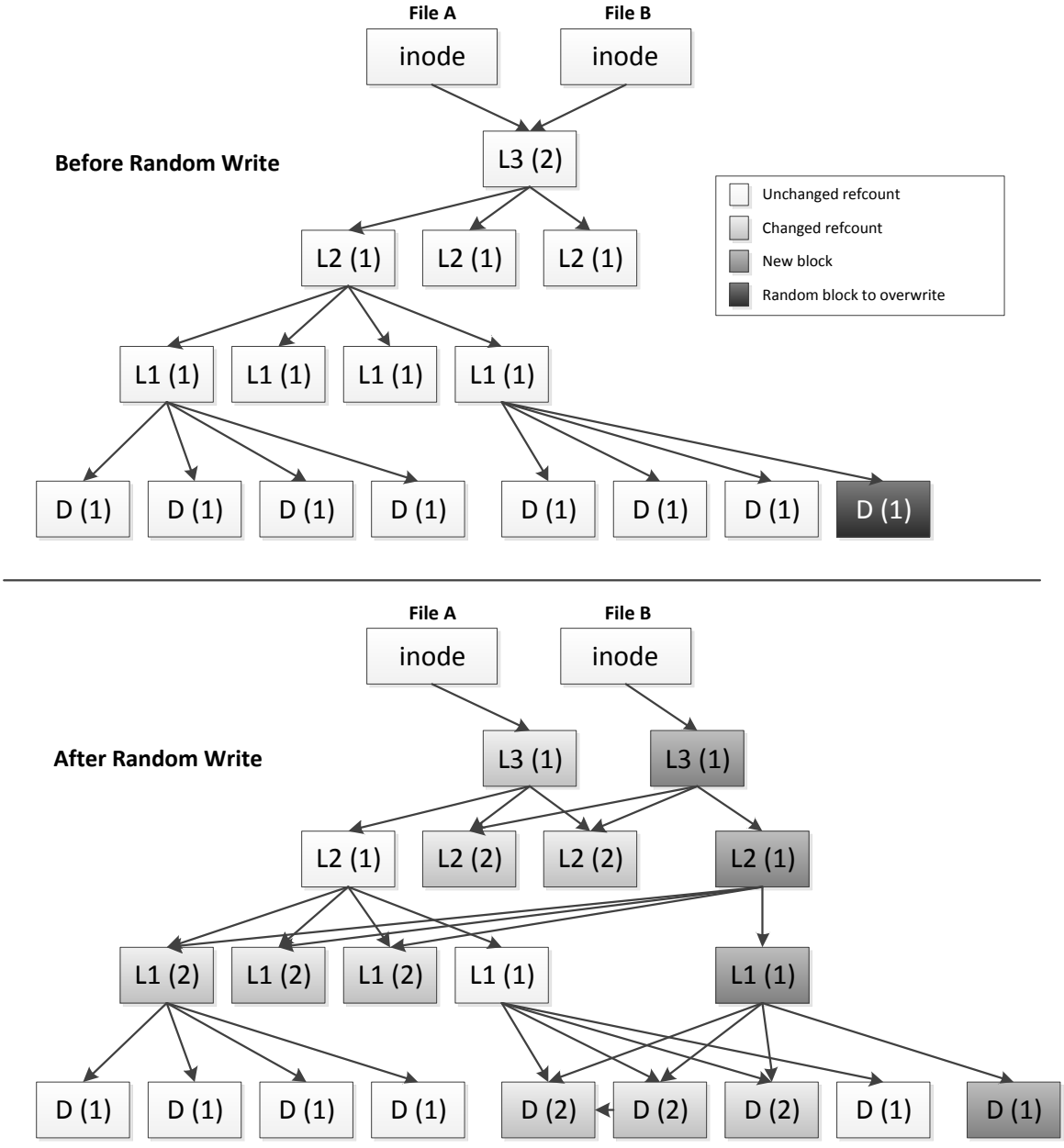


Figure 5-1: Diagram of Structured Reference Counts: Initially, File B is a clone of File A. A user makes a single random write to the black block. This requires updates to the reference counts on all light gray blocks and the creation of the new dark gray blocks.

Figure 5-1 shows how all blocks, even metadata blocks, have reference counts. This makes it possible to share higher level blocks, allowing multiple files to share entire sub-trees of files, instead of just data blocks.

The number of required reference count updates in this scheme can be quite substantial. Consider a 1TB file. In a system with 4KB blocks, this file would have over 256 million data blocks and over 500,000 indirect blocks organized in three levels. A single random write in this file would cause over 2,000 reference count updates if the structured reference count system were used. If a user were to make several thousand small updates to this file, many millions of reference count updates would be required. While reference counting is often the canonical approach for block sharing, its inability to scale efficiently makes it impractical for the types of sharing this thesis seeks to enable.

5.2 Owner Nodes

A second approach is to create new file system objects called *S-nodes*, which would principally store the kind of sharing associated with each set of file system blocks. These additional nodes keep track of "types of sharing" (i.e. block sharing between File 1 and File 2). As files are cloned or blocks shared, S-nodes are created or split to track which blocks are shared and which are unique. There would also be a volume level map (the *S-map*) which tracks which S-node is associated with each file system block.

Figure 5-2 describes this kind of sharing in terms of two files. In **Figure 5-2**, three S-nodes are hierarchically arranged to express the sharing of several blocks between the two files. The S-nodes allow for easy identification of which blocks are shared and which are unique to a specific inode by performing a tree traversal of the S-nodes.

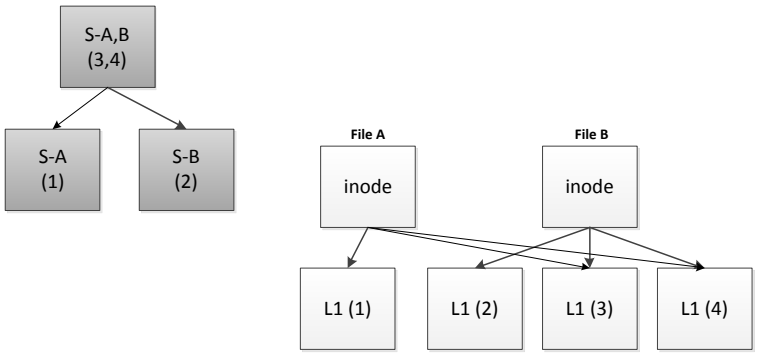


Figure 5-2: Diagram of Owner Nodes: Two files are depicted, both of which share L1 blocks 3 and 4, but each has another unique L1 block. The top level S-node represents the blocks shared between File A and B, while the child S-nodes represent the blocks unique to each.

This approach would require slightly more implementation work because a new class of metadata would need to be created and updated which each file operation. This would require making this new type of object and modifying the write allocator to either create or split these nodes on each operation. There would also be a significant amount of upgrade work involved in bringing forward older versions of WAFL to use this new scheme. This approach has the upside that none of the existing data or metadata blocks need to be updated, meaning that existing data on system could be upgraded with a single metadata tree scan, which would create the needed S-nodes. Also, a new map would need to be created to track which S-node is linked to each inode in the file system.

Unfortunately, the amount of new required metadata may be prohibitive. The inodes on a volume already consume a significant amount of space. In the worst case, this would require several new S-nodes for every existing inode. Consider the case in which every file shares some blocks with every other file. While this is highly unlikely in practice, it would represent an exponential amount of new metadata. While the approach is a possible candidate on systems with a large amount of consolidated block sharing, it is not clear how it would perform on a typical customer storage system. For this reason, this approach was not pursued for this thesis.

5.3 Space Maker

A third solution is to eliminate reference counts entirely, relying on a garbage collection algorithm to reclaim blocks. Instead of explicitly tracking the sharing between two blocks, a block can arbitrarily point to any other block without the need to update reference counts or sharing nodes. Clearly, this has the advantage of accelerating common file operations since the process of linking or unlinking to another block becomes an $O(1)$ operation. **Figure 5-3** depicts several files using the same blocks without the need for reference counts. Furthermore, it is no longer necessary to explicitly free blocks from the active map upon file deletion.

The large upfront performance win does, however, create new problems. If no limits are placed on sharing, then it is impossible to determine whether a block is in use without

scanning the entire metadata block tree. This scheme, Space Maker, is named for the background daemon which would have to reclaim unused blocks. Without any optimizations, this process would seemingly suffer from performance bottlenecks even on relatively small systems, let alone systems with aggregates using tens of terabytes of storage. Furthermore, it is impossible to give an exact accounting of space usage at any point in time. In this approach, the best available counts of space usage would only be an upper bound.

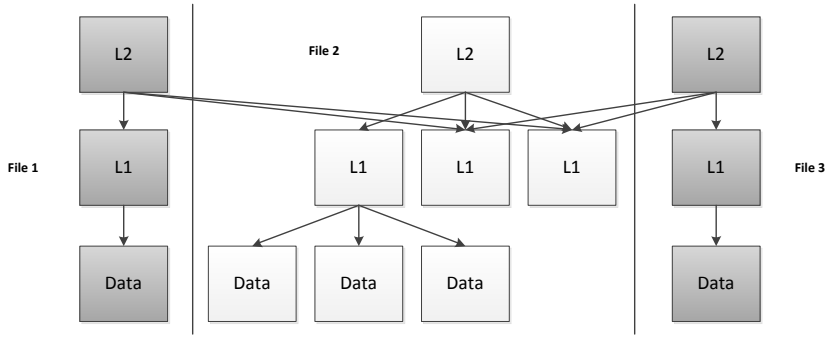


Figure 5-3: Diagram of Space Maker. No reference counts are included since they are not tracked in this approach. Instead, File 1 and File 3 are free to use File 2's blocks without updating any metadata.

Fortunately, several optimizations make this scheme much more desirable in practice. First, the parallelizable nature of this background scanner means that refreshing the active map is not as expensive as a simple linear scan of all file blocks. Secondly, if limits can be imposed on the amount of sharing between blocks, the performance of the scanner becomes much more predictable and useable in an enterprise customer storage environment.

5.4 Summary of Approaches

All of these approaches attempt to create a single block sharing method for existing features that makes it easier to implement future functionality. Each approach has distinct advantages and disadvantages that require additional consideration. Upon examining the workloads of NetApp's filers, the Space Maker approach was determined to hold the most promise of accelerating file operations while supporting the desired levels of new functionality. Specifically, Space Maker performs well in all cases except

when a filer is low on disk space and experiences high IO throughput, a situation which is reasonably rare in most enterprise storage deployments. **Table 5-1** summarizes the qualitative expected performance of a filer using Space Maker under different workloads.

Table 5-1: Anticipated performance under Space Maker.

	Low IO Throughput	**High IO Throughput**
Low Free Space	Good/Acceptable	Acceptable/Marginal
High Free Space	Excellent	Good/Acceptable

It is believed that the vast majority of NetApp customers would experience either excellent or good performance under the Space Maker scheme. These somewhat abstract performance targets essentially express the belief that these customers will experience little to no impact on file operations. It is also believed that sufficient optimizations may produce better than acceptable performance in the worst case (low free space with high IO throughput).

Both because of the potential performance improvements and the simplicity of sharing, the Space Maker approach was selected as the target project of this thesis. Overall, this approach presented the most promise for accomplishing the desired goals.

Chapter 6

Implementation

Implementing the Space Maker prototype required a variety of changes throughout the file system. These changes can be grouped into two categories: file system changes and garbage collection.

6.1 File System Modifications

As described in Chapter 5, the Space Maker approach moves a significant amount of work from the front-end of the file system to a back-end garbage collector. The modifications to the file system involve removing reference counts and simplifying the block free path. More specifically, these changes include:

- Modifying the many block sharing users throughout the file system to avoid updating reference counts.
- Removing all current free path code in the file system (only the Space Maker daemon processes can "free" blocks).
    - WAFL usually creates "zombie" inodes to keep track of inodes and blocks waiting to be deleted. Under the Space Maker approach, every code path that used these structures needed to be modified to avoid zombie creation.
    - Since only a few fixed inode block counters need to be updated on a delete, deletes essentially become a constant time operation under the Space Maker approach.
- Adding performance monitoring instrumentation to make measurements of prototype performance easier.

6.2 Garbage Collection

At the heart of the Space Maker approach is a garbage collector, so a major part of the implementation involved designing a garbage collection algorithm to periodically scan the file system in search of new blocks which will be made available to the write allocator. This was accomplished by writing a file system scanner which marks blocks as in-use in a new copy of the active map, referred to as the *next active map*. At any point in time, the file system maintains the current and next active map.

Additionally, some other structures used by WAFL to track space usage and block allocation in physical regions of the disk are tracked and updated by the Space Maker scanner. Some structures maintained by other parts WAFL are invalidated by the scanner and need to be rebuilt after it completes. These include several caching data structures used by the write allocator to efficiently select future locations to write blocks.

Most of the coding for the Space Maker is internalized inside of the scanner process. To start this process, a single `start_spacemaker` API was created. This can be initiated by the user at the console of a filer, so long as the user has the "Diagnostic" privilege level; as the name suggests, the diagnostic privilege level is not intended for standard customer use. If extended beyond the prototype, this scanner could be scheduled to run at predicable intervals or activated as part of a space management policy when a volume's disk space is low, so a customer would never have to invoke this process manually.

While it might seem that Space Maker could have difficulty on overly full volumes or aggregates, those customers would also need to have high IO throughput needs. Generally speaking, NetApp customers with high IO throughput requirements prefer to have a sufficiently large free space buffer because of the no-overwrite nature of WAFL. It is therefore believed that this scenario is an edge case. While this scenario was considered, it did not serve as a design basis configuration.

Figure 6-1 depicts the operation of the Space Maker scanner. The scanner proceeds by loading each inode in the file system and the associated buffer trees. It is not necessary to load the data blocks into memory, though it is required to load all indirect blocks. With the indirect blocks loaded, it is possible to mark all used blocks in the next version of the active map. It is also important to note that any writes which occur while the scanner is running are also reflected in the next active map. Without this property, it may be possible for writes to be lost between scanner passes.

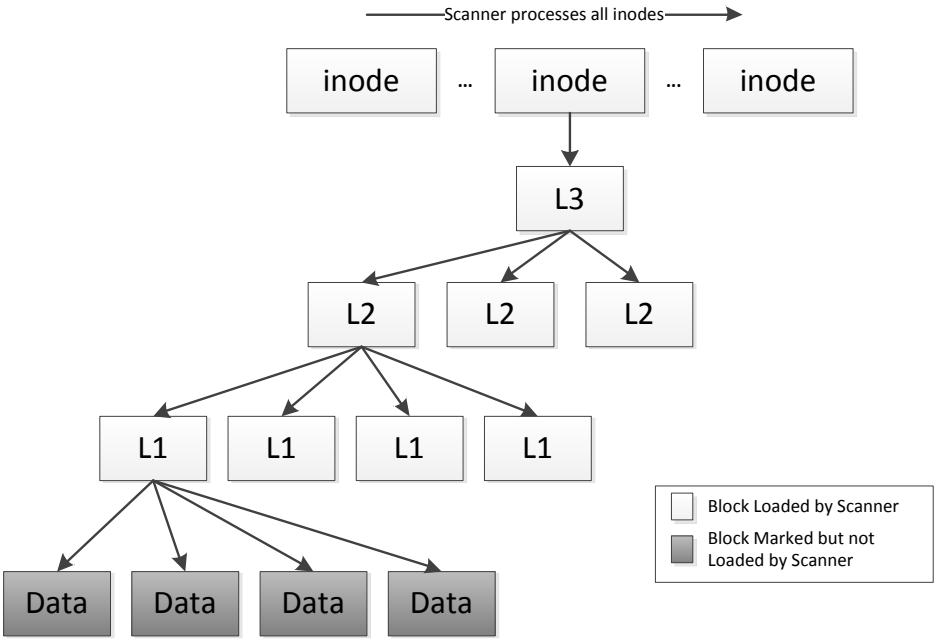


Figure 6-1: Space Maker scanner operation.

Table 6-1 highlights the general operation of the scanner. The scanner performs a complete file system scan by processing all of the inodes and buffer trees associated with every file. Because this functionality was implemented as a *WAFL Scanner*, a common file-system processing apparatus within WAFL, several basic API functions needed to be implemented. This included `start`, `step`, `abort`, and `complete` functions. When the scanner completes, it dispatches a message to the main WAFL message loop to indicate that the next active map is ready to be installed.

Table 6-1: Required scanner functions.

Function	Purpose
Start/Init	• That start method initializes the scanner data structures and schedules the initiation of the scan. • Initialize relevant scan parameters.
Scan Step	• Steps scan progress by selecting the next inode to scan (i.e. the next in-use inode). • Perform single inode scan. • Parses buffer tree associated with inode and marks bits for used blocks in the new active map.
Abort	• Processes a scan abort message and destroys all scanner data structures.
Complete	• Installs the new active map (processing is complete). This will send a WAFL message which will instruct the file system that a new active map is ready to be copied.

6.3 Issues Encountered

A variety of issues were encountered during development that altered the course of the research. First and foremost, the complexity of trying to fundamentally change the architecture of a mature file system like WAFL proved to be exceedingly difficult. While some things like modifying the delete path were manageable, other tasks, like modifying the write allocator proved to be quite challenging. The original plan for the Space Maker involved implementing some of the ideas discussed in Chapter 8. While these ideas were not ultimately implemented, the utility of the Space Maker idea was still made clear, even in the limited prototype designed for this thesis.

Chapter 7

Results and Discussion

The Space Maker approach is a radical departure from previous NetApp file system designs. While some operations are considerably faster, they key was ensuring that others perform nearly exactly the same as a baseline system. The Space Maker approach yields several nice properties that are unique compared to other file systems. Among these are fast delete performance and truly constant-time file clones.

For the prototype of Space Maker to be considered a success, several performance targets needed to be met:

- Write performance of the prototype should be nearly identical to that of a baseline system.
- The time required to delete/unlink a file should be nearly eliminated from a user's perspective. Ideally, the time required to unlink a set of large set of files should be dominated by the network latency required to send the requests.
- The time required to run the Space Maker scanner should be low. More specifically, a targeted time of fewer than 5s per GB of data blocks would be ideal, keeping in mind that Space Maker does not directly read data blocks but rather the indirect blocks that reference them.

In order to assess the performance of the prototype, several benchmarks, described in the following sections, were used.

7.1 Benchmarks

Several micro-benchmarks were used to measure relative performance of several NFS operations and of the prototype system overall.

7.1.1 Evaluation Setup

The baseline performance data was measured on a NetApp FAS3240 appliance with SATA disks using a debug build of an old NetApp software release customized with additional instrumentation. Because of the specialized nature of this release, measurements collected on the baseline system are only comparable to measurements for the Space Maker release. This appliance was chosen because it is a mid-range device with high-capacity SATA disks, a configuration typical of the kind of user likely to benefit from using the Space Maker.

To control for variations in RAID performance, all benchmarks were chosen to fit in the space available on a single physical disk, and tests were conducted on single disk aggregates. This helped to isolate the performance of Space Maker from other factors in the system.

7.1.2 Basic Write and Delete Performance

The first benchmark uses a simple Python script (see **Figure 7-1**) run from a dedicated UNIX-based client to measure file create/write and file deletion/unlink performance of the Space Maker prototype compared to a baseline system.

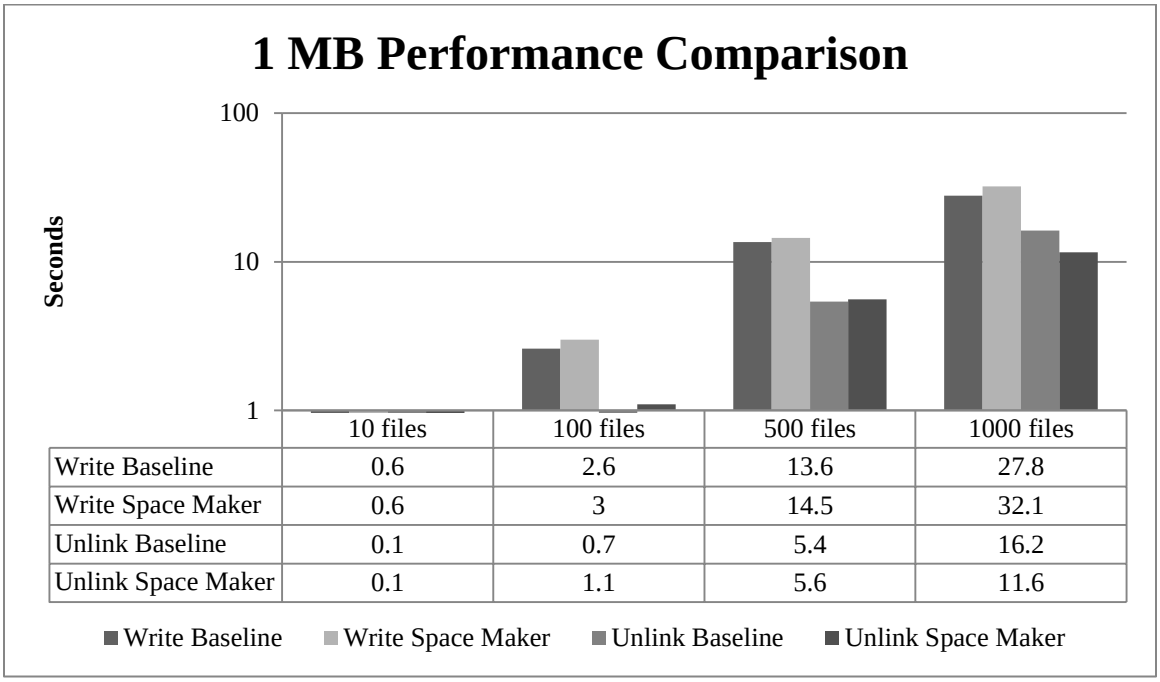
1. File creation time. This is not necessarily the time required to finish writing each file to disk, but it does represent the time required for the NFS client to be notified that the request is complete. This will be measured by recording the time of the Python script on the dedicated client. This time should be roughly the same for both the baseline and the Space Maker systems.
2. File deletion time. This benchmark measures the time to remove all of the files created in the initial Python script. Since the use of the Space Maker heavily optimizes the delete path, there should be significant improvements in the delete time.

```
i = 0
while i < count:
        mkfile file i of some size
        i += 1

i = 0
while i < count:
        unlink file i
        i += 1
```

Figure 7-1: Python pseudocode for basic write and unlink test.

Figure 7-2 compares the performance of the baseline system with the prototype when writing and deleting 1 MB files. While there is a negligible increase in write latency that is more apparent for small numbers of files, there is a noticeable speedup in delete performance with large numbers of files.



	10 files	100 files	500 files	1000 files
Write Baseline	0.6	2.6	13.6	27.8
Write Space Maker	0.6	3	14.5	32.1
Unlink Baseline	0.1	0.7	5.4	16.2
Unlink Space Maker	0.1	1.1	5.6	11.6

Figure 7-2: 1 MB Baseline and Space Maker Performance Comparison

These same trends continue in **Figure 7-3** and **Figure 7-4**, which represent the same benchmark on 10 MB and 100 MB files respectively. As file sizes increase, it is increasingly apparent that write performance is consistent between the baseline and prototype systems while delete performance improves in the prototype. Except in the

case where only a few very small files are being deleted, Space Maker always outperforms the baseline system.

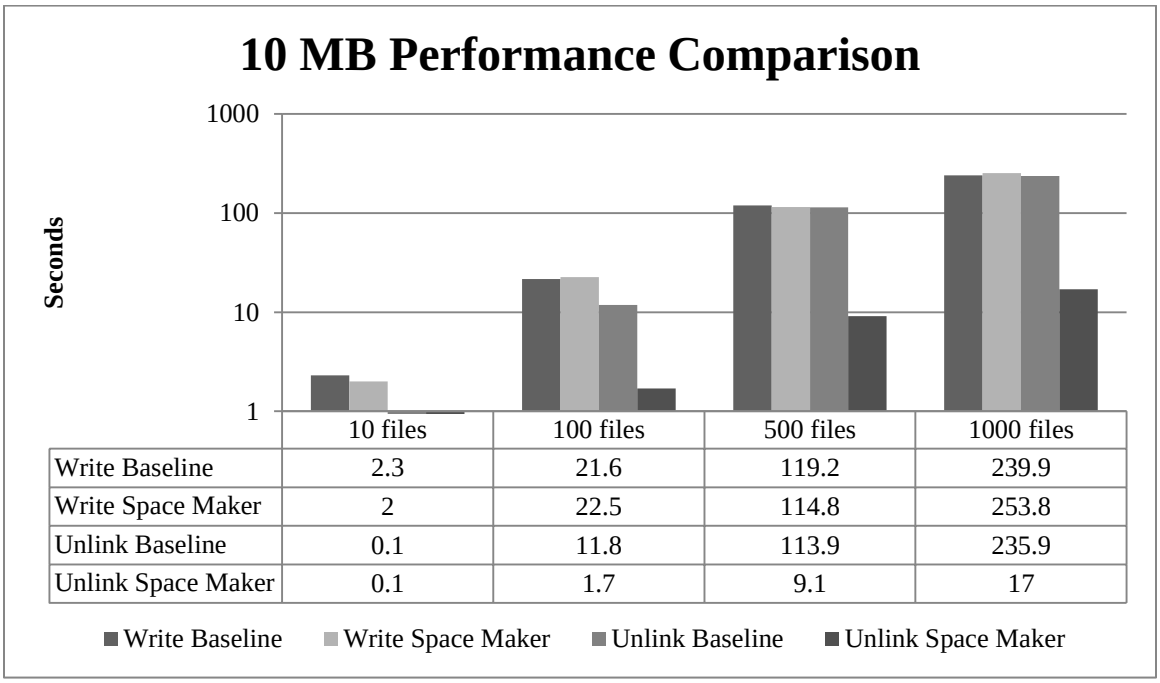


Figure 7-3: 10 MB Baseline and Space Maker Performance Comparison

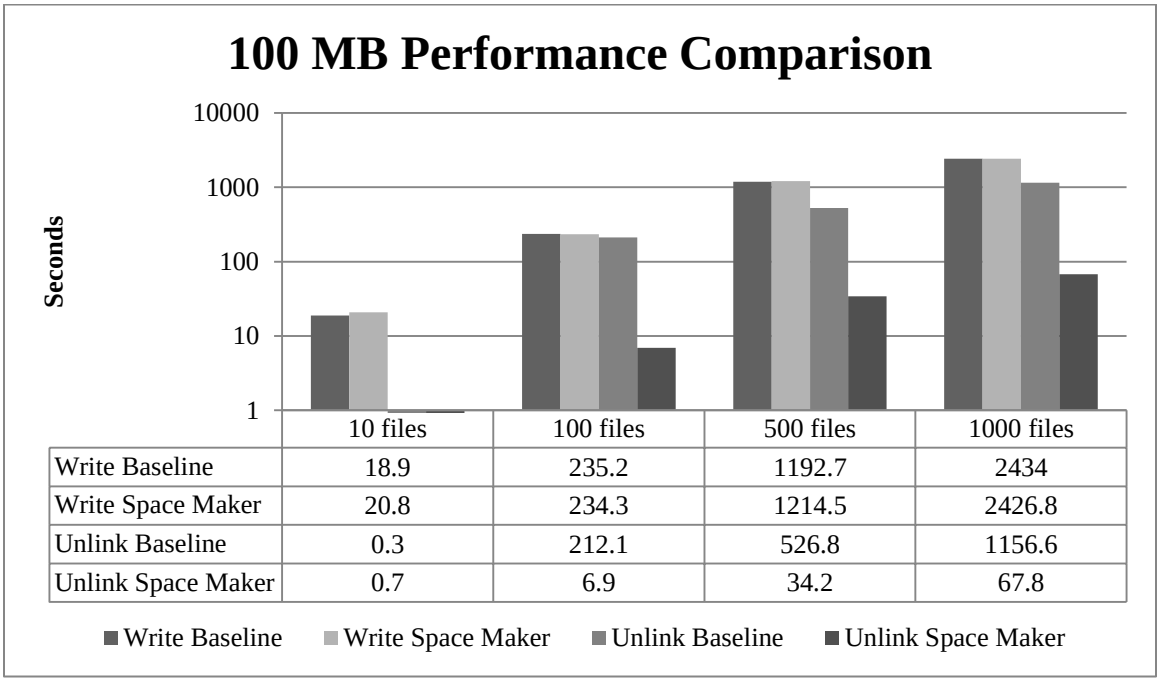


Figure 7-4: 100 MB Baseline and Space Maker Performance Comparison

In the few small files case, Space Maker performance is nearly identical to the baseline system, since network latency is the overriding factor. As the size and number of files increases, it is clear that network latency dominates delete cost. This is especially apparent in **Figure 7-4**.

As was initially expected, implementing the Space Maker machinery had a negligible impact on file create/write performance. Also, as predicted, the Space Maker approach significantly improves delete/unlink performance compared to traditional systems. This performance benefit is much more apparent as file sizes get larger.

7.1.3 Scan Time

Another important performance characteristic of the Space Maker prototype is the time required to run a single pass of the scanner process. While there is some fixed cost to starting the scanner, it performs well with many large or small files. **Table 7-1** summarizes the performance of the scanner under a variety of workloads. In each case, the runtime is largely determined by the time needed to load the each file's indirect blocks.

Table 7-1: Prototype Full-Disk Scan Performance

Space Maker Prototype Scan Performance			
File Set	1000 files @ 1 MB each	1000 files @ 10 MB each	1000 files @ 100 MB each
Data Size	1 GB	10 GB	100 GB
Total Blocks Scanned	272493	2582743	25772774
Scan Time	4s	18s	193s
Time per GB of Data	4s	1.8s	1.9s

7.2 Summary

Even though not all of the original ideas intended for the prototype were implemented, it is clear there is value in the Space Maker approach. Not only is write performance maintained, but delete speed is significantly improved. Also, because of the architecture of Space Maker, implementing new features that want to share blocks is trivial. Constant time full-file and sub-file clones, file de-duplication, and snapshots at every layer of the file system are significantly simplified under the Space Maker design.

From the results of this thesis, it is clear that there are some workloads where Space Maker is ideally suited. These include users with a reasonable amount of free space and a desire to utilize advanced block sharing features. For these users, not only is block sharing greatly simplified, but there is also a performance benefit for scenarios that involve frequently deleting files.

However, it is also clear that there are some workloads where Space Maker may be less ideally suited. Most notably, this includes users who have a low amount of free disk space and also frequently overwrite files. It is similarly clear that users who have no need to use block sharing features would not see an appreciable benefit to using the Space Maker approach.

Chapter 8

Future Work

While the Space Maker idea holds a great deal of promise, especially for a certain segment of storage users, it has a way to go before it can be generally useful. Several approaches were explored during the course of this research to improve performance, though none of these approaches were successfully implemented due to time and complexity constraints.

8.1 Root Sets

As originally envisioned, the Space Maker approach would use the concept of *root sets*, which attempt to partition segments of the disk into logical sharing regions. Root sets can either be mapped to PVBNs (physical volume block numbers) or VVBNs (virtual volume block numbers), which is the convention used by volumes in WAFL to distinguish logical from physical block numbers. This idea has its origin in standard mark-and-sweep garbage collection algorithms which make use of root sets to break code into segments. This provides a bound on the amount of sharing that can take place, making it more feasible to reclaim space. Under this scheme, the Space Maker can focus on separate logical regions, releasing that area's active map once its scan is complete. **Figure 8-1** shows an example disk with root sets.

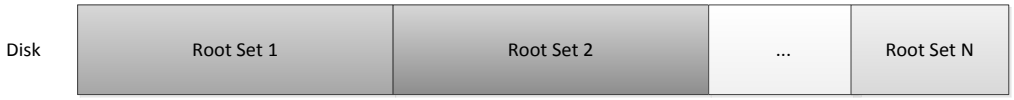


Figure 8-1: Diagram of Root Sets: A set of N root rests will be located on the disk. Root sets are contiguous even though volumes may not be.

Because the Space Maker has a large component which runs in the background, it is imperative that this component run as efficiently as possible. The garbage collection approach with root sets has the advantage that disk scanning is localized to specific

regions. This is important because it would then be possible to release certain parts of the new active map when the scanner has completed processing the associated root set; localizing the cleanup prevents having to scan the entire file system when only a few hot-spots of activity exist. Also, because WAFL is now multithreaded, it would be necessary to integrate with WAFL's multithreading architecture. While some parts of WAFL still run serially, the blocking nature of Space Maker necessitates the use of multithreading. It is believed that these cleanup operations can run at a volume level affinity as long as a root set is entirely contained in a volume, though it may be necessary to run these operations at the aggregate level since volumes do not necessarily consume contiguous regions in the RAID array.

Initial thinking would suggest that free blocks need to be clustered in order to guarantee good performance in the Space Maker scheme. This is because it is necessary for the scanner to focus on regions likely to provide free space, instead of scanning regions without changes in order to keep up with high-throughput systems. However, it is not clear if this will always yield optimal results. There is an underlying assumption that regions with more deletes are more likely to yield more free space after a scanner pass. This may not be true in systems with a lot of sharing.

Root sets were explored during the research for this thesis, though limits in the WAFL file system made it very difficult to guarantee specific blocks are located within a specific disk region.

8.2 Scanner Parallelism

The scanner process implemented for this thesis is single threaded. Since the target benchmarks used a single disk, this had little performance impact, though in high disk count system, it is very likely that the scanner processing will be a performance bottleneck. The scanner is trivially parallelizable since inodes can be processed independently.

Chapter 9

Conclusion

The goal of this thesis was to determine the viability of using a system-wide garbage collector in a general purpose file system in order to facilitate better performance and easier block sharing. To that extent, the results are a success. While only a prototype was implemented, it was shown that delete performance can be improved without affecting write performance. Additionally, the scanner process used for garbage collection was shown to have good performance under a variety of workloads. Using the Space Maker approach, block sharing between full-file and sub-file clones, snapshots, and file de-duplication is greatly simplified. While future study is still needed to realize the full potential of Space Maker, the viability of a file system garbage collector, at least under certain workloads, is now clear.

References

[1] Edwards, J. K., & et al. (2008). FlexVol: Flexible, Efficient File Volume Virtualization in WAFL. *2008 USENIX Annual Technical Conference* (pp. 129-142). USENIX Association.

[2] Hitz, D., Lau, J., & Malcolm, M. (2002). *File System Design for an NFS File Server Appliance.* Sunnyvale, CA: Network Appliance Inc.

[3] Macko, P., Seltzer, M., & Smith, K. A. (2010). Tracking Back References in a Write-Anywhere File System. *USENIX Conference on File and Storage Technologies.*

[4] Patterson, H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S., & Owara, S. (2002). SnapMirror®: File System Based Asynchronous Mirroring for Disaster Recovery. *Proceedings of the FAST 2002 Conference on File and Storage Technologies.* Sunnyvale, CA: Network Appliance, Inc.

[5] Rosenblum, M., & Ousterhout, J. K. (1992). The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*.