

EXPLOITING FILE SYSTEM AWARENESS FOR IMPROVEMENTS TO
STORAGE VIRTUALIZATION

by

Vivek Lakshmanan

A research paper submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2009 by Vivek Lakshmanan

Abstract

Exploiting File System Awareness for Improvements to Storage Virtualization

Vivek Lakshmanan

Master of Science

Graduate Department of Computer Science

University of Toronto

2009

File systems are tasked with storing, organizing, and retrieving valuable data for long periods of time. This requires them to provide excellent fault-tolerance and reliability standards throughout their extended lifetime. As a result, most file systems follow a conservative development model and evolve slowly. In comparison, the underlying storage hardware, as well as the requirements of the modern storage environment are changing much more rapidly. Due to the growing adoption of Storage Virtualization, hardware configurations have grown in complexity, but the file system is shielded from the details by the block interface. This puts file systems at a disadvantage when trying to make efficient use of the underlying hardware and in general, acts as an impediment to meeting higher-level goals for the storage stack entirely from within the file system. At the same time, the block interface also obfuscates most of the vital file system context from the hardware, limiting its ability to compensate for the file system's limitations and make sound decisions for the entire storage system. We show that simple hints from the file system exposed to a cognisant block layer can be effective in achieving high-level goals for the storage system without requiring functional modifications to existing components of the storage stack, like the file system itself or the block interface. We demonstrate the viability of our approach by achieving significantly improved corruption detection and failure recovery performance without causing a discernible slowdown in most normal file system workloads. In fact, we sped-up certain metadata-heavy workloads by upto 60%.

Contents

1	Introduction	1
1.1	Research Problem	3
1.2	Possible Solutions	3
1.3	Proposal	4
1.4	Contributions	7
1.5	Chapter Overview	8
2	Background and Motivation	9
2.1	FSCK	9
2.1.1	Effect of ext3 File System Layout on fsck	12
3	Design	14
3.1	Design Goals	15
3.2	Major Components	15
3.3	File System Hints	16
3.3.1	Granularity	16
3.3.2	Explicit Hints	16
3.3.3	Inferred Hints	17
3.4	Remapping Mechanism	17
3.4.1	Remapping Data Structure	18
3.4.2	Exploiting Journaling Support	21

3.4.3	Failure Model	22
3.4.4	Failure Recovery	23
3.4.5	Preventing Lock-In	24
3.5	Remapping Policy	24
4	Prototype Implementation	26
4.1	Remapping Mechanism	27
4.1.1	Overhead	28
4.2	Changes to the ext3 File System	29
4.2.1	Exposing Metadata Hints	29
4.2.2	File System Recovery	31
4.3	Modifications to the NBD Protocol	32
5	Evaluation	33
5.1	Evaluation Platform	33
5.2	Evaluation Methodology	34
5.2.1	Benchmarks	34
5.2.2	Choosing Remapping Target Device	36
5.3	Online Performance	36
5.3.1	Custom Microbenchmark	39
5.4	FSCK Performance	39
5.5	Discussion	40
6	Related Work	43
6.1	Information Gap in Storage Systems	43
6.1.1	Smarter File Systems	44
6.1.2	Smarter Block Layer	45
6.1.3	Improved Cooperation Between File System and Block Layer	46
6.2	Storage Virtualization	47

6.3	File System Consistency Check Performance	48
7	Future Work	50
7.1	Short Term	50
7.2	Longer Term	51
7.2.1	Integrating Solid State Disks in Existing Storage Systems	51
7.2.2	Generic Block Level Remapping Mechanism	51
8	Conclusion	52
	Bibliography	53

List of Tables

4.1	Overhead Per Remap Region Segment with 8192 Blocks	28
-----	--	----

List of Figures

1.1 top	2
2.1 top	10
2.2 top	11
4.1 top	27
4.2 top	28
5.1 bottom	37
5.2 bottom	37
5.3 bottom	38
5.4 bottom	39

Chapter 1

Introduction

The traditional storage stack is a hierarchical design (see Figure 1.1). The file system is usually the component of the operating system tasked with organizing, storing, and retrieving data. It interacts with an abstract representation of the underlying hardware, known as the block layer, through a thin interface. And the block layer in-turn, interacts directly with the underlying hardware. File systems are expected to store persistent data reliably for long periods of time. They are expected to provide excellent fault-tolerance and various self-healing capabilities. Due to their sensitive nature, most file systems go through lengthy development and testing cycles, making for an inherently conservative development model that is highly resistant to change. In comparison, hardware configurations, hardware specifications, as well as the requirements from storage systems, all change much more rapidly.

We briefly survey a few of the changes in the storage landscape in the last decade to provide some context:

Changes in Hardware Configuration Several new approaches to tailor storage systems to the needs of the deployment have emerged in recent years. Storage Virtualization has been one of the key reasons why this has been possible. Storage Virtualization refers to the ability to abstract away the physical location of data from its identity. It exploits the abstraction provided by the block layer to integrate emerging technologies and ca-

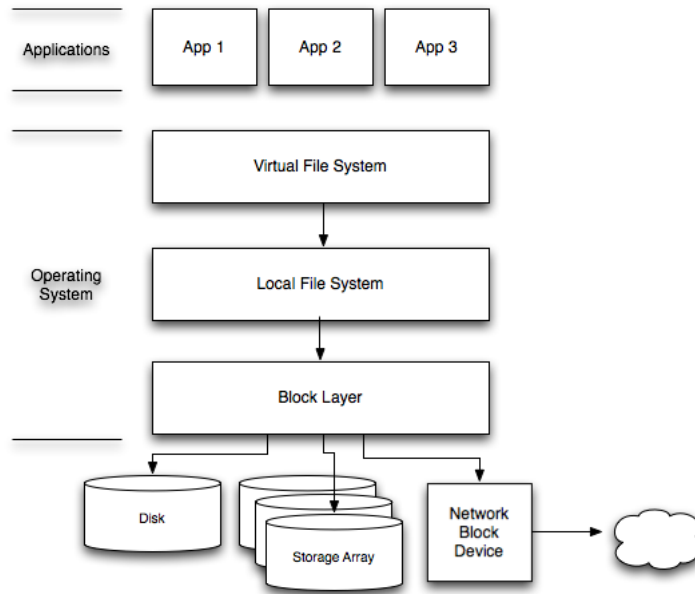


Figure 1.1: A *high-level overview of the modern storage stack*.

pabilities into existing hardware configurations. For instance, Volume Management on commodity operating systems allows users to merge several backing disks into a single logical view and provide advanced features like software RAID [17, 9]. Similarly, Storage Area Networks (SANs) scale storage virtualization to levels necessary for adoption by enterprises. SANs allow the collective storage capacity of the environment to be pooled and multiplexed for several clients through high-speed interconnects. With the recent popularity of whole-system virtualization and live-migration, SANs provide uninterrupted access to the data even as the client instance migrates across the network. Another network storage technology is iSCSI, which offers similar features without the high initial cost of a dedicated Fibre channel network by leveraging commodity interconnects.

Changes in Hardware Characteristics Traditional storage systems have relied on the assumption that disks fail in a fail-stop manner. However, it is only recently that the failure-models of storage hardware are being properly understood. For instance, recent studies

have shown increasing rates of latent sector errors and signs of transient failures in the storage stack [4]. Another phenomena is the mixing of fundamentally different storage hardware within the same storage system to leverage differences in the performance characteristics and costs. For instance, storage systems for HPC environments often use slower archival media like tapes for long term storage, but disks are used to cache recently accessed data. More recently, emergence of new storage media like Flash, with very different performance, reliability, and power-usage have prompted debate over the ideal approach for integrating them in modern storage systems [8].

Changes in User Requirements Priorities for deployments might change over time. For instance, recent years have seen power consumption become a significant concern for storage systems.

1.1 Research Problem

Rapid changes, as described above, in storage hardware, combined with the changing demands of modern storage deployments are in conflict with stability and reliability goals of the file system. As a result, file systems may not be able to make effective use of storage hardware to meet higher-level goals, whether related to performance, reliability, or power.

Our work is aimed at improving the interaction between existing file systems and the underlying storage hardware to meet higher-level goals for the storage stack without compromising their stability and reliability, nor requiring modifications to the standard block interface.

1.2 Possible Solutions

One solution is to expose the details of the entire storage stack to the file system. With this information, the file system can be designed to implement complex reliability, failure recovery, and power conservation policies. There are two challenges that make this approach impractical:

- As previously mentioned, file systems are designed with stability and reliability in mind. Adding such features would significantly increase the possibility of bugs in the file system, which carries the risk of permanent data loss. Even relatively simple file systems with years of active testing and development have been shown to suffer from a number of serious bugs [43, 4]. Most modern file systems are rewrite-in-place, e.g. NTFS and Ext3, that maintain a static on-disk layout for compatibility. This dramatically limits their dynamism.
- Secondly, the block layer interface is extremely thin and does not allow the file system to glean information about the underlying hardware easily. As a result, a number of interesting black-box techniques have been proposed to determine certain hardware characteristics. For instance, Talagala et al. proposed micro-benchmarks that could be used to estimate low-level disk geometry, and latencies [37]. Some file systems use similar techniques to determine RAID stripe widths, etc. [10]. However, as storage virtualization increases in complexity, such mechanisms will not be as effective.

We feel that modifying the file system to cater to new developments in the storage stack is fraught with danger, and in some cases, impractical.

1.3 Proposal

The more practical alternative in our opinion is to optimize the storage layer for the hosted file system. However, the block interface obfuscates file system context from the storage layer. Our approach is to expose file system hints to the storage layer so that the overall goals of the storage stack can be met. This approach has been recognized before by others as being powerful and practical. For instance, Semantically Smart Disks leverage in-depth understanding of rewrite-in-place file systems like ext2 to infer file system operations, and are able to provide powerful features such as secure deletion and journaling for file systems that don't support it [33]. We consider this form of black-box inferencing promising, but we choose a similar approach of

providing hints directly to the storage layer. The challenge we face is that while the hints provided to the storage layer need to be flexible enough to allow powerful policies to be crafted, they should not require a departure from the existing block interface.

These hints and corresponding policies allow the block layer to address a large number of the problems associated with Storage Virtualization. For instance, even the interaction between the file system and a storage array, one of the simplest forms of Storage Virtualization, is littered with problems and missed opportunities for offering powerful new features due to the lack of visibility across the block layer:

Performance Without accurate knowledge of the RAID parameters the file system may make block allocation decisions that causes related data to straddle disk stripes, which can lead to poor I/O performance. Stein showed that in several cases a random block allocation policy was less sensitive to changes in storage virtualization parameters like the RAID stripe size, and in fact, performed better than the complicated block allocation heuristics of a popular local file system [36]. Another problem is the possible concentration of crucial metadata on a single disk. This can not only reduce the effective parallelism available to the storage system, but also dramatically increase the possibility and impact of failure in the overburdened disk.

Reliability and Failure Recovery Ordinary RAID arrays redistribute stripes across disks transparently. With little insight into the complexity of the underlying storage hardware, the file system is unable to implement replication or block allocation strategies to provide variable fault tolerance capabilities for data.

Power Conservation Given the scale of modern storage systems, there is heightened interest in scaling down power consumption dramatically. For similar reasons as above, the file system is unable to adapt its I/O requests for avoiding spun-down disks or exploiting power efficient hardware like Flash [22].

However, with the help of hints from the file system, the storage array could avoid placing

blocks likely to be accessed together across RAID stripes. This would be quite simple for example, if the file system informed the block layer of related blocks during block allocation. If the storage array could distinguish data from metadata, it could detect inordinately high traffic for metadata being directed to a single disk and take corrective action. With additional insight, such as the identity of related blocks, the storage array would be able to implement interesting replication policies. For instance, frequently accessed files could be replicated within a disk, as well as striped across an array to allow graceful degradation beyond the failures tolerable by the RAID configuration [32]. Important metadata could be mirrored or checksummed as per the user's fault tolerance requirements [14]. Lastly, the storage array can be designed to distribute blocks to enforce a desired level of power savings. This can be achieved either by redirecting I/O to lower-power storage hardware like Flash, or by redirecting requests targeted at spun-down disks to active ones [22].

In this work, we wish to demonstrate that hints from the file system can be exploited to meet overall goals of the storage stack at the block layer. In particular, we demonstrate how simple hints from the file system can be used to address the longstanding challenge of improving File System Consistency Check (*fsck*) performance for a popular rewrite-in-place file system, Ext3, without requiring any functional modifications to the file system that may jeopardize its reliability.

File systems like Ext3 scatter metadata across the disk to reduce seeks to the corresponding data and improve online performance. However, with the increase in main memory capacity, most metadata remains in cache once accessed, making the motivation for this design decision mostly obsolete. However, *fsck*, the file system consistency checker for Ext3, requires traversing all the metadata blocks in the disk, and is adversely affected by this decision. As volume sizes increase, *fsck* time has been consistently increasing. Though file systems like Ext3 use journaling to reduce the need for *fsck* runs to recover from unexpected crashes, *fsck* is still recommended after recovering from a disk failure. Moreover, with the increasing concerns of transient errors in the storage stack, frequent *fsck* checks become an easy mechanism to scrub

the disks for inconsistencies. However, the performance cost of fsck is prohibitively high for even moderate-sized volumes at the moment.

We aim to demonstrate the viability of our approach by improving fsck performance at the block layer by exploiting simple hints from the file system. We assume that consumption of additional disk space or requiring additional volumes is an acceptable price to pay for reduced failure recovery times.

Our solution has been designed with the following guidelines in mind:

1. The file system should expose the hints to the storage layer with minimal modifications
2. The hints must be simple enough to not require modifications to the standard block interface. At the same time, these hints must provide enough information to the storage layer so that the desired goals can be achieved.
3. Existing fault-tolerance and recovery guarantees provided by the file system must not be violated.
4. Online performance must not suffer unduly.

1.4 Contributions

We make the following contributions in this work:

- We show how even stable file systems can benefit from providing hints to the block layer. In particular, we demonstrate how this approach can be used to improve failure recovery times by improving the performance of fsck, and other metadata heavy workloads for existing file systems.
- We provide a generic remapping mechanism at the block layer that allows modifying the block layout on the underlying hardware arbitrarily, while isolating the file system from such changes.

1.5 Chapter Overview

We begin by providing an extended motivation for choosing improving failure recovery time as an important goal for the storage stack in Chapter 2. Chapter 3 provides a detailed design of our approach, while Chapter 4 provides details of our prototype. We quantify the impact of our approach by evaluating its performance in Chapter 5. Chapter 6 provides an overview of the existing work in this area. We provide a sampling of other improvements to the storage stack our approach enables, and that we wish to pursue in the future in Chapter 7. Lastly, we present our conclusions in Chapter 8.

Chapter 2

Background and Motivation

Our research is focussed on exploring how file system hints can be used by the storage layer to meet the overall goals of the storage stack. In this thesis, we focus on a single goal: improving the time to detect and recover from file system corruption. In particular, we demonstrate how file system consistency check (fsck) performance can be improved using our approach. The rest of this chapter provides insights into the problems existing block layouts pose to fast file system consistency checks.

2.1 FSCK

Users expect their persistent data to be available and consistent at all times. As a result, storage hardware and software is expected to meet very high standards of reliability and fault tolerance. File systems, being an integral component of the storage stack, are some of the most meticulously developed subsystems of operating systems. Nonetheless, their fault-tolerance characteristics are constantly being tested in the wild as hardware deployments grow in complexity. Even relatively simple, stable file systems have proved to be vulnerable when faced with the emerging threats of transient I/O failures [15, 43]. Journaling [38] and soft-updates [12] are two online techniques that have been adopted by the file system community at large for providing quick recovery from unexpected crashes. However, they do not provide adequate

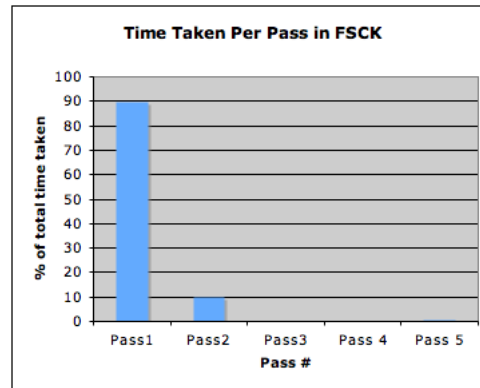


Figure 2.1: Contribution to total *fsck* time made by each phase for a run on a 40 GB disk at 91% utilization containing mostly MP3s. Pass 1 corresponds to verifying all blocks are accounted for in each file. Pass 2 traverses directory entries to verify they are valid. Pass 3 checks that the root directory of the file system can be reached from every valid directory. Pass 4 verifies that link counts are accurate for valid files. Finally, Pass 5 verifies that accounting information, like used/free space, block counts, etc. for the file system are accurate.

protection against such problems. This is because they assume the integrity of unmodified sections of the file system image. Silent data corruption and bitrot can affect the integrity of the stored data and not be detected by the mechanisms mentioned above. *fsck* provides an easy and ubiquitous way to verify the integrity of the storage system.

A significant problem with file system corruption is the ease with which it can propagate across the image. Bairavasundaram et. al. used type-aware fault injection to demonstrate that even stable commercial-grade file systems can propagate corruption and lead to catastrophic data loss [4]. Frequent runs of *fsck* can reduce the time to detect inconsistencies, and avoid widespread corruption propagation throughout the on-disk file system image. However, this ubiquitous recovery mechanism is not exercised often due to its poor performance. For instance, it takes over 2 hours for a simple consistency check on a 33% filled ext3 file system on a 1.7 TB volume. This time grows much larger based on how spread-out the metadata blocks are on disk. Some large-scale storage systems require days, not hours, for completing a consistency check, tilting the balance in favour of restoring from a tape backup and losing some

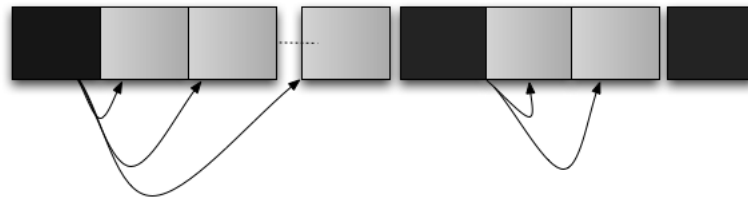


Figure 2.2: *Block layout on the Ext3 file system. Metadata blocks (darker) point to data blocks (lighter) and are placed next to the data they point to.*

data, rather than taking the system offline and performing a lengthy file system consistency check [40].

Fsck performs its checks in a number of passes. The first iterates through inodes to identify allocated blocks and verify file sizes. The second verifies the directory structure. The third verifies directory connectivity with the file system’s root directory. The fourth verifies reference counts, and the fifth checks if summary information is accurate. The first two passes are most time consuming for fsck since they deal with traversing the metadata blocks on disk, as shown in figure 2.1. The problem is caused by dynamically allocated metadata blocks. To improve read performance, they are allocated close to the data they point to. As a result, during consistency checks, the disk spends a large amount of time seeking from one metadata location to another while reading only a small fraction of all the blocks on disk. This is due to a design decision made by early file systems when main memory was not sufficient to avoid accessing the disk for metadata-heavy and sequential workloads. However, the increase in primary memory size over the years has resulted in most metadata blocks being cached in the operating system buffer cache, thus rendering this heuristic redundant for maintaining good sequential read performance. Next, we examine how block layout affects fsck performance in ext3 in detail.

2.1.1 Effect of ext3 File System Layout on fsck

The ext3 file system was inspired by the Fast File System (FFS). Like FFS, it splits the disk into logical segments called block groups (known as cluster groups in FFS). This design allows improved accounting and management of metadata. Inodes, data structures that aggregate information about files, are allocated statically at file system creation¹ and distributed evenly across the block groups. Their allocation status is tracked by bitmaps that are placed close to their corresponding bitmaps. Most of the rest of the blocks are left as unused, free for storing data. However, certain metadata blocks are allocated dynamically so they are close to the corresponding data blocks. We discuss each type of these blocks next:

Indirect Blocks For small files, the inode itself contains pointers to a list of blocks that contain data. However, for larger files, pointers to the on-disk blocks are maintained in metadata called indirect blocks. As the file grows in size, another layer of indirection is created in the form of double-indirect blocks, where each entry in the block is a pointer to an indirect block. Lastly, for really large files, a triple indirect block which contains addresses of double indirect blocks can also be allocated.

Directory Blocks Directory blocks contain name and inode number pairs. These blocks form the hierarchical namespace we traditionally associate with a file system. These blocks need to be accessed to resolve the location of files on disk.

Figure 2.2 shows how metadata is allocated close to the blocks it points to. Since the metadata needs to be accessed first to locate the associated data, the access to the data block can be serviced without requiring additional seeks. As a result, accessing the data pointed-to by these metadata blocks can be done with little to no seeking, while metadata heavy workloads, like fsck, require significant amounts of seeking. If sufficient main memory is available, subsequent accesses to the metadata can be serviced entirely from the buffer cache, but a workload like

¹Support for dynamic inode expansion has been added in recent years but for the sake of simplicity, we restrict ourselves to the canonical ext3 on-disk layout.

fsck must pay a high initial penalty for accessing metadata blocks.

We plan on improving file system consistency check performance by clustering metadata blocks closer together. Normal buffered workloads would not notice a significant performance hit since subsequent accesses can be served from the operating system's cache. However, workloads like fsck which require traversal of most metadata blocks on disk can be sped up dramatically.

Chapter 3

Design

We wish to ensure that the stability and availability goals of the file system don't become an impediment to its ability to keep up with the dynamism of storage hardware, and meet high-level goals for the storage stack.

Our solution is to expose hints from the file system to the block layer which are simple enough to be embedded in existing block interfaces, but at the same time, informative enough to allow powerful policies to be implemented at the block layer. To improve file system consistency check performance, the block layer needs to be able to distinguish between metadata and data blocks. Given the hierarchical nature of the storage stack, the block layer does not have enough exposure to file system context to do so in a generic manner. Instead, we propose the file system expose hints to the storage layer directly.

This chapter provides a detailed description of our design. We begin by stating the primary design goals of the system, followed by a discussion of the key components of the system. We then discuss file system hints and the related challenges to finding the right granularity and medium to expose these to the block layer. We then provide the design of the remapping mechanism in detail. Lastly, we briefly discuss the remapping policy used in this work and possible enhancements in the future.

3.1 Design Goals

Our solution has been designed with the following guidelines in mind:

1. The file system should expose hints to the storage layer with minimal modifications
2. The hints must be simple enough to not require modifications to the standard block interface. At the same time, these hints must provide enough information to the storage layer so that the desired goals can be achieved.
3. Existing fault-tolerance and recovery guarantees provided by the file system must not be violated.
4. Online performance must not suffer unduly.

3.2 Major Components

Our proposed solution for improving fsck performance leverages file system hints through two main components:

Storage Mechanism The block layer can use the file system hints to initiate various storage mechanisms. In this work we implemented a block remapper which maps a block to a different region on disk or an entirely different device, and makes this mapping persistent. Additional storage mechanisms can be added to act upon file system hints for various purposes. For instance, infrastructure for replicating and checksumming selected blocks could be added [14].

Storage Policy Policies allow the system to use the hints provided by the file system and the storage mechanisms provided by the storage layer to achieve high-level goals. In our case, the policy remaps metadata blocks and clusters them closer together in order to reduce failure recovery time.

3.3 File System Hints

File system hints expose some parts of the file system’s context to the block layer. Finding the correct granularity and the ideal delivery mechanism to expose these hints to the block layer was an important consideration for us to meet the design goals stated above. We discuss these in greater detail here.

3.3.1 Granularity

Our aim is to ensure that the hints are generic enough to avoid over-specialization for a specific file system, and specific enough to allow administrators to implement interesting policies.

For our goal of clustering metadata blocks, we simply require the file system to differentiate metadata blocks from data blocks. However, more complex schemes are feasible and would allow more powerful policies. For instance, exposing relationships between blocks could further improve clustering. This could be achieved by identifying blocks associated with the same file, or the same user. For dynamically allocated metadata, exposing their deallocation can help the block layer minimize space overhead for our remapping mechanism. In addition, this could enable more complex features like secure deletion [31, 30].

3.3.2 Explicit Hints

It is possible to use fields in existing block interfaces to expose hints to the block layer. For instance, Meisner et. al. have recently proposed using the group ID field in the SCSI command set to provide differentiated service to workloads on hybrid storage systems *XXX: need citation*. In this approach, hints are passed in-band, along with I/O requests, and the block layer can implement its policies by intercepting and acting upon them synchronously. This is the model we primarily adopt because it requires minimal changes to existing block-level interfaces and

simplifies the implementation of the storage mechanism ¹. An alternative would have been for the file system to expose hints to the block layer asynchronously but we don't discuss this approach here.

3.3.3 Inferred Hints

Most current file systems follow a largely static block layout. The block layer can be made aware of this layout and infer information about file system operations by simply doing range checks on I/O request locations to implement some policies. The body of work on Semantically Smart Disks has explored this approach in detail and also sheds light on the complexity of inferring high-level file system operations through this approach [33]. However, we do leverage some well-known semantics of our target file system, Ext3, to design our remapping mechanism, as described in section 3.4.1. Though the same information - i.e. the layout and size of block groups could have been sent to the file system through an explicit hint, we decided against this approach since this information rarely changes between different file system images of Ext3.

3.4 Remapping Mechanism

The remapping mechanism allows clustering of blocks. We allow blocks to be remapped to arbitrary locations in the remap disk, providing significant flexibility for implementing clustering policies at the storage layer. However, as a result, the remapping information must be recorded, persisted, and kept consistent even in the presence of system failures. As a result, the storage layer must maintain a persistent remapping data structure.

¹Our prototype uses the Network Block Device which doesn't natively support such features which required a slight modification to Linux's NBD protocol, see the next Chapter for details.

3.4.1 Remapping Data Structure

The data structure used to maintain remapping information is crucial to maintaining fast online performance, because it must be traversed to identify whether a block is remapped or not, and the location of a remapped block. A Radix Tree is a common data structure for storing this information. This makes sense when any logical block accessed by the file system might be remapped. However, for improving fsck performance, we can restrict ourselves to just the metadata blocks. Metadata only constitutes a small portion of the blocks on disk. As a result, the effective remappings possible are only a small subset of the blocks on disk. We adopted the Inverted Page Table (IPT) as our remapping data structure, since it is used for storing page tables on 64 bit architectures where the effective physical memory is only a fraction of the overall 64 bit address space. Support for different remapping data structures can be added as needed for other storage policies. In addition to the IPT, we maintain a bitmap which tracks the allocation status of the blocks in the remap region.

We split the remap region into equal length segments. Each segment manages the metadata for a single block group of the base file system. There are several reasons for this design. First, the target file system, Ext3, performs coarse grained clustering by preferring to keep related metadata within the same block group. For instance the inode allocator in Ext3 tries to allocate new inodes such that the files in the same directory are stored in the same blockgroup, close to the directory. Our approach can automatically leverage clustering of metadata in the remap region. Second, by having each segment maintain its IPT and bitmap, we avoid unnecessary contention on a global radix tree during peak loads. Block level remapping solutions which do not exploit file system hints, like [20], are susceptible to this problem. This can be considered an example of where our design exploits inferred hints, since the block layer exploits knowledge of the file system without hints directly from it.

Remapping Mechanism Requirements

Our remapping mechanism tries to provide good online performance with reasonable overhead, while upholding the consistency semantics of the file system. Below, we summarize how these requirements are met by our design.

Consistency Recall that we aim to uphold the existing fault tolerance and reliability properties of the file system. However, by introducing additional metadata at the block layer, it is possible that the block layer's state can become inconsistent with that of the file system. For instance, if the file system issues a request that results in a remapping, and the block layer acknowledges the request without making all intermediate modifications to the IPT blocks persistent on disk, a failure could result in the data being inaccessible. This problem can occur even if we exploit the file system's support for journaling, as explained in Section 3.4.2. Given that we remap file system metadata blocks, the file system could become grossly inconsistent. To avoid this situation, we use delayed acknowledgements, i.e. we make sure the block layer never acknowledges a write request until all metadata associated with it is on disk as well. This provides the semantics that the file system expects off a regular disk and the consistency requirements of the file system are maintained.

Performance Fast block resolution can be implemented by caching most of the remapping information in memory. For a 1.7 TB file system, the storage layer would need roughly 2 GB of RAM to maintain the remapping information entirely in memory. Given the specifications of modern high-end storage arrays, this overhead is manageable and can be tailored to the deployment by tuning the size of remap segments to correspond to the data-metadata ratio in the file system. It is possible in some cases for the IPT chains to grow long, but, in our experience, this has not been an issue, especially given that the time for accessing the IPT is easily dwarfed by disk access latency.

However, delaying acknowledgements may effect performance negatively. This is because

for each write request that requires an update to the remapping information, the segment's bitmap, a HAT block, and upto 2 IPT blocks may be required. To improve I/O performance, the delayed acknowledgements approach does not impose any ordering on the remap information blocks it flushes to disk, as long as all the dependent block are written to disk before it acknowledges the I/O request to the file system.

Note that the overhead for flushing the remap blocks to disk only needs to be paid when the remapping information is updated. Read and writes to remapped blocks which do not require updates to the remapping information can be serviced directly without incurring the above overhead. Therefore, only metadata block allocation and deallocations incur an overhead. We implemented a number of performance optimizations that further reduce the impact of delayed acknowledgements.

First, we allow multiple outstanding requests to the block layer. This provides the potential of batching together updates to the same remapping blocks. This allows us to trade latency for throughput. A similar approach was used in Parallax [20]. An alternative would have been to implement a secondary journal in the storage layer. Instead of delaying acknowledgement of requests that require an update to the remapping data structures till all dependant blocks have been written to disk, we could simply ensure that the block and a corresponding entry in the log representing the source and destination of the new remapping is persisted before acknowledging the request. This would reduce the latency of block remapping requests. However, this will require an additional journal write, and a journal region, increasing the space overhead in the remap region. We have currently adopted the former approach, though a journal based approach can also be supported if necessary. For instance, if the file system does not support journaling, the latency of properly remapping metadata can be reduced by using a journal in the remapping layer.

The other major performance optimization involves exploiting the journaling support available most modern file systems to convert the remapping of metadata blocks into an asynchronous operation so that it is no longer on the critical write path, while upholding the relia-

bility guarantees provided by the journaling file system. We discuss this optimization in greater detail in the next Section.

3.4.2 Exploiting Journaling Support

Most modern file systems have adopted journaling to provide a degree of protection against inconsistency due to unexpected failures. We have already discussed in Section 2.1 why simply relying on journaling or soft-updates is not sufficient and a full file system consistency check continues to be relevant. However, we exploit journaling to reduce the effective performance overhead of adding (or removing) block remappings due to delayed acknowledgments.

The standard journaling support on Ext3 maintains a hidden file on disk where metadata blocks are journaled. Periodically the contents of the journal are checkpointed to their actual location on disk and the head of the journal is updated. To minimize the online performance impact due to delayed acknowledgements, our hints identifying metadata blocks are exposed to the block layer only during journal checkpointing, which is a background task, while journal updates which are in the critical path are allowed to pass-through as regular data writes. Exploiting journal checkpointing for passing hints to the block layer also offers the added advantage that by the time journal checkpointing is initiated, short-lived metadata allocations have likely already been purged. This can happen with workloads that create and delete files rapidly, for example compilation workloads, which avoids added overhead.

Note that the support for journaling does not obviate the need for delayed acknowledgements. However, the delay is attached to the asynchronous journal write-back and not the original write. Even though the journal can replay committed but un-checkpointed transactions, we must make sure all intermediate block updates in the remapping data structure are written to disk before acknowledging the original request. If the checkpoint requests are acknowledged too early, the file system could complete the journal transaction and erase its journal entries before the block-layer's remapping information has been flushed to disk, and a system failure would result in the data from the just completed transaction being irrecoverably lost!

Journal write-back is not entirely free. In Ext3, journaled buffers are pinned in main memory till their checkpoint is complete. As a result, metadata blocks requiring updates to the remapping information in the block layer will cause increase memory consumption during journal commits. Another factor that can influence performance is the frequency with which the journal contents are checkpointed. This is dependant on a number of variables. For instance, if the free space in the journal reaches a threshold, or there is limited main memory left, the journal write-back thread is activated. There is also a static timeout that triggers the journal write-back periodically. Nonetheless, delayed acknowledgements during checkpointing, we are able to reduce the effective performance overhead of metadata clustering at the block layer sufficiently.

There has been previous work that has tried to exploit journaling for improving file system reliability from the block layer. Please refer to 6.

3.4.3 Failure Model

Hardware Failure One of the benefits of our remapping approach is that we can use either faster or more fault tolerant hardware for the remapping region, providing improved performance or fault tolerance for metadata operations. The remapped blocks are just as vulnerable to transient faults in the storage stack as they are in the original file system. Since the goal of our current system is to improve consistency check performance, we do not attempt to prevent possible corruption of these blocks and assume the file system consistency checker will do the needful. However, it is possible to establish a complimentary goal of improving fault tolerance for metadata blocks by implementing additional policies which may replicate or checksum remapped blocks in a similar manner to I/O Shepherding [14].

In the event of power failure or similar occurrences, the file system recovery mechanism should restore the system to the same state as an unmodified one. We discuss failure recovery in Section 3.4.4.

Software Bugs Since we leverage existing file systems, we are not immune to bugs in their implementation. However, by minimizing changes to the file system, we guarantee their effect is no different on our storage system than on any other existing deployment. Moreover, we refrain from making any modifications to the file system consistency checker, which should restore our system to the same functional state as an unmodified one. Bugs in our block layer implementation may be an additional source of vulnerability, however, we argue that since our remapping mechanism is relatively simple, it is easier to test and verify than an entire file system.

3.4.4 Failure Recovery

So far we have seen how our remapping mechanism works under normal conditions. We will now discuss the approach we used for recovering from failures.

Consistency with File System State

Assuming the file system supports journaling, all committed but not yet checkpointed transactions would be replayed during file system recovery. The file system's hinting support should embed the metadata vs. data hints in its journal entries. In our remapping mechanism, adding or removing remapping entries is idempotent, provided the relative ordering of the operations is maintained, which the journaling mechanism ensures [38]. As a result, when the journal is replayed, previously acknowledged block remapping operations still in the journal transaction will be replayed in sequential order, thus ensuring the remapping information will be consistent with the state of the file system by the end of journal recovery.

Internal Consistency

Though consistency with file system state can be reached by replaying the journal during recovery, the remapping information may not be entirely internally consistent. In particular, some space in the remapping region may be wasted. For instance, if a failure occurs while the storage

server is flushing the IPT table to disk, some blocks, such as the IPT's bitmap may be written to disk but the corresponding updates to the IPT blocks may not have been flushed. As a result, a block in the remap region is lost. This inconsistency can be reconciled by performing a quick traversal of the remapping information to *garbage collect* unused references as it is retrieved from disk at load time. Since only a small amount of remapping information is required per segment and all of it is clustered together, this garbage collection can be done very quickly. We have not yet implemented such a recovery mechanism in our current prototype, and leave it for the near future. It is important to note that the lack of the garbage collector does not affect correctness in the presence of a journaling file system.

3.4.5 Preventing Lock-In

One problem that plagues storage systems is that adoption of a particular file system or block layer solution forces the user to keep using the chosen solution. Our block-level remapping mechanism has an additional feature that allows users to stop our system partially or completely. For instance, suppose the file system is no longer able to provide hints to the block layer. Our remapping mechanism will then treat all incoming requests as pass through and no longer allocate or deallocate metadata blocks. At the same time, previously remapped blocks will continue to be accessible, as long as our block layer solution continues to be used. However, if one wishes to detach a file system entirely from our system, our remapping information allows rewriting the remapped blocks to their home location on the base file system. the actively remapped blocks can be rewritten to their home location on the base file system. Once completed, the base file system can be mounted directly, with all its changes entirely intact.

3.5 Remapping Policy

The remapping policy for improving consistency check performance dictates that metadata blocks be remapped to a separate region and clustered closer together, thereby dramatically

cutting down on disk seeks experienced by fsck. The remapping policy controls how the metadata blocks are clustered. Despite the fact that hints provided by the file system simply separate metadata blocks from data blocks, the remapping policy can still be fairly influential. For instance, more sophisticated approaches can be based on access patterns for improving performance of online metadata-heavy workloads, or based on known access-patterns for improving fsck. With additional hints, such as the type of metadata blocks, or additional context from the file system, more interesting policies can be implemented. We discuss some of these ideas in Chapter 7.

We will now take a closer look at the implementation of our prototype.

Chapter 4

Prototype Implementation

A logical view of our prototype can be seen in Figure 4.1. As previously mentioned, our base file system is Ext3 - a popular rewrite-in-place file system actively used in a number of commercial deployments. The file system interacts with the block layer which eventually passes block requests to a pseudo-block device called the Network Block Device (NBD) [39] available as part of the Linux Kernel. The Network Block Device converts block requests from the file system to a custom protocol over TCP to a server which acts upon the requests. The NBD server interacts with disks through the raw pseudo file system exposed in Linux and in order to bypass the buffer cache, it uses direct I/O for all its I/O requests. Our block remapping mechanism lies primarily in user space in the server while a modified version of the NBD module and the ext3 file system reside in the client. We chose to use this split design because it made prototyping, experimentation, and evaluation easier. Our server has been implemented on top of *Akash*, a storage server built to study cache partitioning in shared storage environments [35].

In this chapter we discuss the intricacies of our prototype implementation. We begin by discussing the remapping datastructure introduced in Section 3.4.1 in greater detail. We then describe the changes we made to the Ext3 file system to pass hints to the block layer. Finally, we present the changes required to the NBD protocol to pass file system hints between the

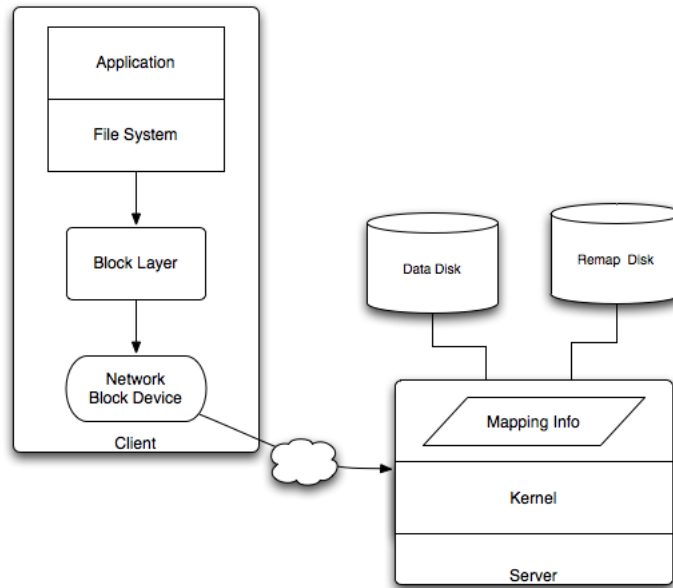


Figure 4.1: *This diagram shows the current prototype. The file system and the NBD module on the client have been modified so that hints can be passed over the network to the server which hosts the remapping module and the backing disks.*

client and the server.

4.1 Remapping Mechanism

The design of the IPT data structure can be seen in Figure 4.2. Given the LBN for an incoming request, it is hashed into the Hash Anchor Table (HAT) which contains the head of a chain of IPT entries. Each IPT entry consists of the LBN of the target block in the remap region, the LBN of the source block in the base file system, a pointer to the next IPT entry in the chain, and an additional 4 bytes currently left as spare. The IPT entry list is traversed to resolve the incoming request to an LBN in the remap region. Metadata block allocations result in additions to the end of the IPT chain. Similarly, when dynamically allocated metadata like indirect and directory blocks are deallocated, the corresponding remappings can be removed. One natural optimization we would like to pursue for the future is a small cache, which stores the last few

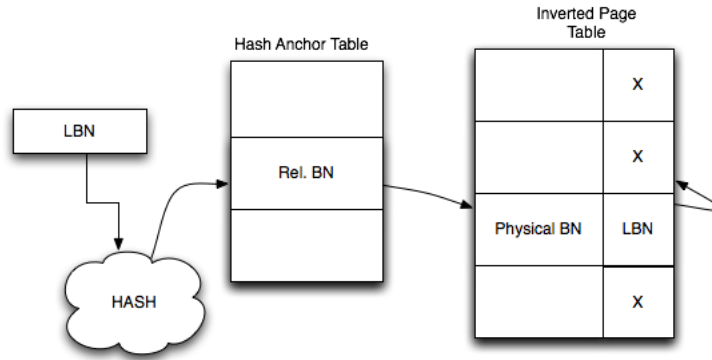


Figure 4.2: *Physical block resolution on an Inverted Page Table given a Logical Block Number (LBN).*

Table 4.1: Overhead Per Remap Region Segment with 8192 Blocks

Block Type	Number of Blocks
Hash Anchor Table	8
IPT Entries	32
Bitmap	1
Superblock/summary	1

resolved LBNs and their translations to avoid the overhead of the traversing the IPT. However, in our experience, since almost all of the IPT stays in memory in the server, translations are not a source of performance problems when compared to disk I/O.

Our infrastructure addresses blocks using 32-bit logical block numbers (LBNs) which allows us to address up to 16 TBs of data in the base file system, given 4 KB blocks. This corresponds to the limits of a standard ext3 file system at the moment.

4.1.1 Overhead

The overhead associated with maintaining the remapping metadata can be seen in table 4.1. It currently amounts to 42 blocks per 8192 remapped blocks in the remap region - an overhead of roughly 0.5%. Note that we expect metadata blocks to be only a fraction of the number of

blocks in the base file system. In table 4.1 we assume 25% of the blocks in the file system are metadata which is an overestimation for most deployments.

4.2 Changes to the ext3 File System

The ext3 file system does not expose hints to the block layer by default. Our design for improving fsck performance calls on the file system to distinguish metadata blocks from data when issuing write requests. Therefore, minor changes to the ext3 file system have been made so it exposes these hints to the block layer.

4.2.1 Exposing Metadata Hints

As discussed in Section 3.4.2, in order to minimize overhead in the write-path, we only embed our hints in requests corresponding to metadata checkpointing. We do this by annotating a descriptor for metadata block updates. We currently only annotate indirect and directory block updates. The ext3 file system's journaling code wraps the block update with another descriptor pointing it towards a destination in the journal. As a result, our original annotation is obscured and the write request generated for the journal I/O has no annotation. Once the journal transaction for the block has been committed however, the descriptor pointing to the journal is stripped-off and the subsequent checkpointing request would contain the annotation. This block request is passed to the NBD block device which relays them to the storage server along with its annotation.

Recall that metadata like indirect and directory blocks are dynamically allocated. As a result, these can transition from being regular data blocks to metadata and back. Given that we only allocate a fraction of the total volume size for clustering metadata in the remap region, it may be desirable to undo remappings as metadata blocks are deallocated. However, there is a possibility the block is reallocated as metadata shortly thereafter, forcing us to pay the performance penalty for wasted updates to the remapping information. As a result, there is

a space-time tradeoff with accurately tracking deallocations for metadata blocks. In order to keep our approach applicable to a variety of workloads, we decided to track and undo mappings for deallocated metadata and save space in the remapping region. However, the approach we use to hint deallocations to the block layer is different for each type of block.

Indirect Block Deallocation

As far as the file system is concerned, an indirect block is still allocated till the corresponding bit in the block bitmap is unset and written to disk. However, tracking deallocations on this operation would require keeping the last updated copy in memory in the server, and then figuring out what blocks used to be indirect and were recently unset in the bitmap. Instead, we decided to pursue an optimization. In Ext3 deallocation of an indirect block involves zeroing out the indirect block, followed by its removal from an inode, or from another indirect block. However, the entire operation is not committed till the corresponding entry in the bitmap is unset.

The internal journal in Ext3 has a fixed maximum size, of roughly 128 MB. As a result, some transactions, like a large delete can not fit entirely within journal transaction and must be broken up. In order to guarantee consistency despite failures between a delete which is split between two transactions, the deletion must be carried out in strict order. Hence, the indirect block is zeroed in a bottom up manner when deleting or truncating a file.

We indicate deallocation of the indirect block by tagging the checkpointing request that zeros it out, instead of tagging the unset of the bitmap. This approach is convenient for passing hints to the block layer and eases the remapping policy implementation in the block layer. On seeing an indirect block deallocation tag, the block layer removes the existing remapping for the block and redirects the write request to the location requested by the file system to begin with.

Tagging the zeroing of the indirect block as a deallocation also retains correctness since any subsequent reallocation as an indirect block would be tagged as metadata, in response to

which the block layer can update its remapping. If instead, the block is used as ordinary data, no updates to the remapping need to be made. In all cases, the current state of the block can be accessed by read requests correctly.

Directory Block Deallocation

Unlike in the case of indirect blocks, the deletion of a directory may not actually require any updates to them. As a result, hinting directory block deallocation to the block layer is more challenging. Directory blocks are normally deallocated when the directory inode is removed. Although directory deletion requires that it be empty, i.e. the only valid entries in the directory may be either `""` or `..`, the actual deletion requires a process similar to that followed for deallocating an indirect block discussed above. We tag a directory block as being deallocated when it becomes empty. Note that unlike indirect blocks, the file system may not actually deallocate the emptied block in most cases. However, this approach still retains correctness. If the file system does deallocate the directory block after emptying it, our remapping remains consistent. However, if the file system updates it as a directory block, the request would be tagged as a reallocation and the block layer would need to create a fresh remapping and redirect the block to the remap region.

4.2.2 File System Recovery

Our basic block type tags (metadata/data) are maintained in a descriptor attached to the in-memory version of the block that is used for checkpointing by the Linux Kernel. This information is pinned in memory until the checkpoint request is acknowledged by the disk. However, in the case of an unexpected system failure, this in-memory information will be lost. The journal replay during recovery will not have the hint attached and our remapping mechanism will not be able to exploit the lost hints from the file system to implement its policy for the replayed requests.

To avoid this, we needed to make minor modifications to ext3's journaling layer to write

block-type hints as part of the journal transaction to disk. In addition, the recovery mechanism in the file system is correspondingly updated to read the block type tagging from the journal and retransmit the requests to the block layer. Note that this is the only modification we have made that affects any on-disk state. However, unlike the main file system, the journal is transient and does not need to remain consistent across versions of the file system, provided any necessary recovery of left over transactions is completed before an upgrade.

4.3 Modifications to the NBD Protocol

Linux's NBD provides a simple protocol to pass read and write requests. This is functionally equivalent to the most common features in the standard SCSI command set used to talk to disks. However, it does not readily allow passing file system hints to the block layer as the SCSI command set does (see section 3.3.2). As a result, we made minor modifications to the packets used by the NBD to pass some additional tags.

Chapter 5

Evaluation

In this chapter we will try to demonstrate that our approach of combining file system hints with a block-level remapping mechanism can implement powerful policies. Specifically, we show that our approach is effective at improving failure recovery times without requiring fundamental redesigns of parts of the storage stack. In this chapter we briefly discuss our evaluation strategy and some preliminary performance results. We quantify the improvements to fsck through artificially aged file system images and compare against unmodified Ext3. At the same time we demonstrate that our approach imposes very little overhead for some standard file system workloads. We then present a detailed discussion of our evaluation, problems encountered, and possible future improvements.

5.1 Evaluation Platform

Our prototype consists of a Network Block Device server, its corresponding NBD module, and a slightly modified version of the Ext3 file system. The NBD server is implemented entirely in user space and resides on a remote machine with direct access to raw disks, while the client machine needs to install the NBD kernel module and mount the modified Ext3 file system on the NBD pseudo block device. For our evaluations we used two Dell SC-1450 servers with 2 Dual-core Xeon CPUs@3.60 GHz, 2 GB of main memory, a 250 GB SATA disk and a 36

GB SCSI disk respectively. Both are connected to a dedicated 1 Gigabit Ethernet switch. One was used as the server and provided raw block-level storage to the other, which acts as the client. The server is running Debian 5 with the accompanying 2.6.26-2 kernel, while the client is running Debian 4 and a custom built Linux 2.6.23 kernel. We don't explicitly require these versions of the operating system for our prototype but we have not tested on other platforms.

5.2 Evaluation Methodology

File system consistency check performance is highly dependant on the block layout of file system images. It is extremely sensitive to fragmentation and gets considerably worse with the age of the file system. It is a challenge to artificially generate a file system image that exhibits a realistic amount of fragmentation in a compressed amount of time [34]. Moreover, the aging workload must be repeatable in order to make a fair comparison of file system consistency check performance. We experimented with a number of aging alternatives, but settled on *compilebench* due to its repeatability, ease of use, and realistic workload [18]. We describe our benchmarking methodology next.

5.2.1 Benchmarks

In addition to being our aging mechanism, we use *compilebench* as another benchmark. It artificially ages file systems by generating a large number of directories and issuing I/O representative of several iterations of compiling, deleting, and patching the Linux kernel's source trees. We added a flag to the *compilebench* script to prevent it from deleting the created directory structure after a benchmarking run is complete. We run *fsck* on this artificially aged file system to compare *fsck* performance.

Compilebench creates 20 directories of Linux kernel sources initially, followed by 30 operations randomly chosen from the following:

- Additional kernel source tree expansions.

- Patching an existing kernel source tree. This operation results in several files in the directory tree being modified.
- Compiling the kernel source tree. This results in several additional files being created in the directory tree.
- Reading an entire source tree. This exercises data and metadata.
- Cleaning the kernel sources. This selectively deletes files throughout the directory tree.
- Deleting an entire source tree.
- Running *stat* on each file in one of the source trees source trees. This is a typical metadata intensive operation.

The seed for the random number generator to choose an operation is kept constant, therefore subsequent runs result in the same operations being repeated.

We then run custom microbenchmarks proposed by Piernas et al. in [25]. This consists of the following operations:

Read-meta (r-m) : Find files larger than 2 KB in a directory tree. This should only read metadata blocks.

Read-data-meta (r-dm) : Read all regular files in a directory tree. This accesses data and metadata blocks.

Write-meta (w-m) : Create a directory tree with empty files. This only causes writes to metadata blocks.

Write-data-meta (w-dm) : Create a directory tree. This should write to data and metadata blocks.

Read-write-meta (rw-m) : Copy a directory tree with only empty files. This should cause both reads and writes to the metadata blocks.

Read-write-data-meta (rw-dm) : Copy a directory tree containing non-empty files. This should cause reads and writes to both metadata and data.

5.2.2 Choosing Remapping Target Device

Our approach allows dynamically allocated metadata like directory and indirect blocks to be moved to a separate volume - the remapping device. This can be either a separate disk or another partition. Using a separate disk enables additional parallelism, but requires investment in additional hardware. The metadata disk must be considered at least as reliable as the disk containing data since crucial metadata would be stored on it. However, since there are far fewer metadata blocks than data blocks in most file systems, this allows the disk to be much smaller in size. In our evaluation we assume a 1:4 ratio for metadata to data. Thus, the 250 GB SATA disk is partitioned with only the first 95GB formatted with Ext3. We leave the rest for storing the remapping information when evaluating the alternative of placing data and metadata on the same disk but in different partitions, or otherwise, leave it unused. The first 11GB of the SCSI disk store the server's operating system and applications, while the remaining 25GB is used as our remapping target when evaluating performance with separate data and metadata disks. We evaluate both of these alternatives in terms of their online usage and file system consistency check performance.

5.3 Online Performance

It is important to remember that improving file system consistency check performance must not come at the cost of dramatically reduced online performance. As a result, providing comparable online performance is a prerequisite to pursuing any effort that tries to improve fsck performance. At the same time, remapping metadata might also improve performance for some other common metadata workloads. As a result we find it worthwhile to present our online performance evaluation.

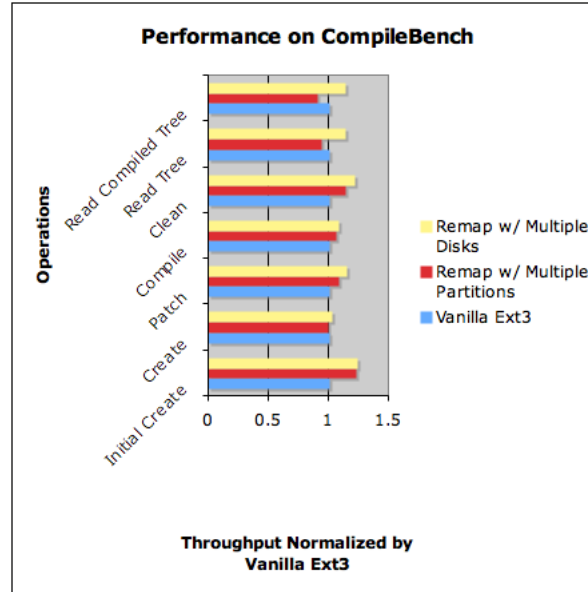


Figure 5.1: Average throughput for compilebench’s operations normalized by Ext3 performance on kernel source trees. Higher values are better.

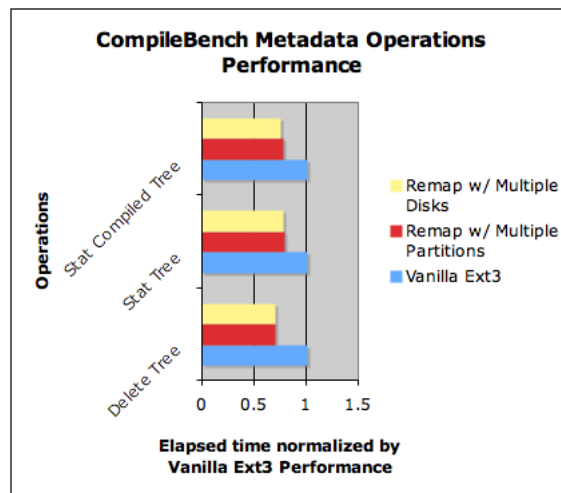


Figure 5.2: Elapsed time for metadata heavy tasks reported by compilebench normalized by Ext3 performance. Note that lower values are better here.

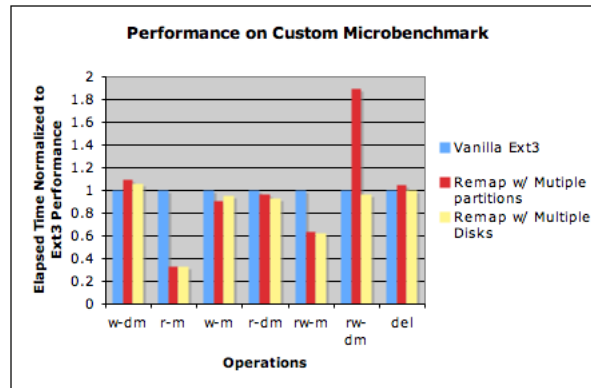


Figure 5.3: *Elapsed time for operations from a custom benchmark described in section 5.2.1 normalized by Ext3 performance. Lower numbers are better here.*

Compilebench Results

We ran compilebench on a freshly formatted Ext3 file system stored on the 95GB partition of the SATA disk as described above. The results from compilebench can be viewed in Figures ?? and 5.2. Compilebench measures elapsed time for each benchmark iteration and tracks the total size of the data set for each benchmark. It reports its results in the form of average throughput for data heavy workloads which is calculated from the known dataset size and elapsed time across all runs. Metadata heavy workloads are reported in elapsed time.

The results show that for most data heavy operations, our approach adds virtually no overhead, and in fact most operations show a slight improvement, even when metadata is located on a partition on the same disk as data. The remapping approach performs particularly well on metadata heavy tasks such as cleaning a kernel source tree, running *stat* operations, and deleting a source tree. The additional disk seems to provide up to a 20% boost for a number of data operations. We attribute this to the additional parallelism the second disk provides. For metadata heavy tasks like *stat* and *delete*, the remapping approach performs significantly better.

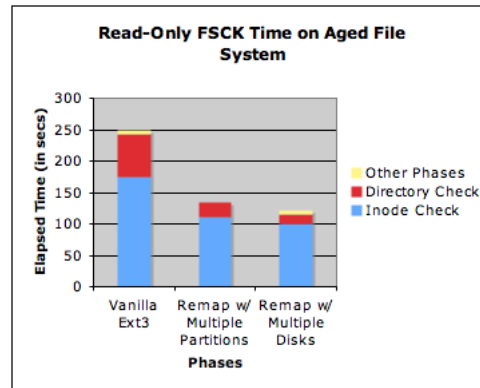


Figure 5.4: *Elapsed time for an fsck consistency verification run in read-only mode. This compares our approach against vanilla Ext3 with metadata on the same partition as data, and on a separate disk.*

5.3.1 Custom Microbenchmark

We ran the custom microbenchmark to better understand online performance of our remapping approach. Its results can be seen in Figure 5.3. For metadata-only workloads, our approach is significantly better, as we see improvements of 40-80%. For operations like the expansion of a kernel source tree that write to data and metadata, we seem to incur a slight overhead of 5-10%. However, for operations that both read and write to data and metadata, the single partition solution performs very poorly. This is because of our decision to only remap dynamically allocated blocks like directory and indirect blocks. Operations that require frequent accesses to inodes still stored in their original location in the data disk incur the overhead of seeking across partitions when traversing from metadata (like a directory entry) to the inode it points to. This overhead disappears with the additional parallelism of a second disk.

5.4 FSCK Performance

We now discuss the results from running fsck on an Ext3 file system on a pass through NBD server and on our prototype with two different configurations: one with both data and metadata

stored on the same disk but in different partitions, and the other where the metadata is stored on a separate disk. Each of the file system images have been aged using the same workload generated by compilebench.

Figure 5.4 shows the results from a fsck run on a file system utilizing roughly 11 GB of the 95GB available in the partition. Verification of file size and accounting for allocated blocks is a dominant contributor to the fsck execution time, followed by verification of directory integrity. Using our approach, even with a single disk we are able to reduce the time required to verify directory integrity by roughly three times, this is further improved by adding an additional disk. However, verifying the integrity of inodes does not show a similar reduction. We once again attribute this to the separation of indirect and directory blocks away from inodes in our system. We provide more insight into the problem in section 5.5.

5.5 Discussion

Our prototype demonstrates that it is possible to leverage simple hints from the file system to implement powerful policies using our block remapping approach. In particular, we demonstrated that we could dramatically reduce failure recovery times for an existing file system without requiring functional changes to it.

Though the results of our evaluation are encouraging, our decision to not remap every metadata block seems to cause some performance problems, particularly when using a single disk with separate partitions. For instance, one of the custom micro-benchmarks tries to copy a large directory tree. This requires metadata and data to be both written and read. Moreover, this operation requires updates to a large number of inode blocks. Recall that the inode table is partitioned into equal-length segments stored in every block group. Since we do not remap inode blocks, operations like directory tree copying require the disk to fetch data from the remap region as well as the data region. When a single disk is used to store both the data and metadata region in separate partitions, this forces the disk to incur lengthy seeks, causing substantial

overhead for such operations. A similar phenomena limits the performance improvements in the inode verification phase of fsck as well.

One approach to reducing this penalty is to simply add an additional disk. In our tests, this was quite successful. However, when this is not possible, remapping inodes could also be considered. Since inodes are not dynamically allocated, the overhead associated with delayed writes for allocation and de-allocation does not apply, thus runtime performance should not suffer. We plan on evaluating this policy in the near future.

We also noticed that the actual number of metadata blocks allocated on our test runs was much smaller than the 25% we had conservatively estimated. We believe metadata and fsck performance can be improved further by tuning the size of our remapping region segments to better reflect the amount of metadata in the file system.

One important note about our evaluation is that we allowed updates from the client to be stored in the buffer cache in the server. Ideally we would have used direct I/O which would replicate the behaviour of modern disks by acknowledging updates only after they have been persisted. As a result, the prototype evaluated above would not provide the same fault-tolerance guarantees as we had designed for.

We decided against using direct I/O because it imposed penalties of up to 10 times on any workload running on the client. We postulate that since the NBD server is multi-threaded, it reorders incoming request streams, which may well be sequential, into random I/O requests. With direct I/O, all requests seemed to pass directly to the disk, without the possibility of coalescing with neighbouring updates. As a result, every workload on the client seemed to exhibit the characteristics of random I/O, thus making it impossible to accurately gauge performance improvements due to our remapping mechanism. With buffered I/O, incoming requests are allowed to coalesce and better utilize the disks. We plan on investigating this problem further and finding an appropriate solution.

To summarize, we feel these initial performance evaluations are satisfactory and encouraging. Our current heuristics for allocating remapped blocks within the metadata disk are

extremely simplistic. This leaves room for further work. One of our immediate goals is to evaluate the fault tolerance and recovery properties of our approach. We would also like to evaluate our approach against other similar approaches like DualFS.

We now discuss the prior work that has influenced our research.

Chapter 6

Related Work

In this work, we have shown how our approach of combining strategically placed hints from file systems, with a simple remapping mechanism, and specifiable policies can be used to meet overall goals of the storage stack. In this chapter we place our work in the context of existing research in storage systems, and in particular, work related to bridging the so-called *information gap* in the storage stack. We also discuss other recent block-level remapping mechanisms. Finally, since we dedicated the assessment of our approach to boosting recovery time from file system failure by improving fsck performance, we discuss some of the other recent efforts in achieving the same goal.

6.1 Information Gap in Storage Systems

We have highlighted previously that the lack of insight across layers in the storage stack, particularly between the file system and the underlying hardware, leads to missed opportunities for performance optimizations or interesting reliability and fault tolerance features *XXX: Add track back to the section where we mentioned this!*. This so-called *information gap* between the file system and the hardware has been a popular area of research and has been approached from various directions. However, one of three themes is dominant in most work in this space: improved insight for file systems to the underlying hardware, more exposure for hardware to

the file system above it, or an intermediate approach suggesting greater cooperation. We summarize some of this work here.

6.1.1 Smarter File Systems

Some have argued that file systems should be able to adapt to the characteristics of the underlying hardware to improve performance. For instance, Schindler et al. argued that the file system should be provided details of the underlying hardware so that the file system's block allocator can optimize the block layout [28]. Others have proposed black-box techniques for inferring hardware characteristics which can be used to achieve similar goals [37, 41]. We feel the pace of advances in storage hardware poses a challenge to such an approach. Since file systems follow a slow and methodical development model due to their sensitive nature, they are likely to lag significantly behind the state of the hardware, thus being unable to exploit its features appropriately. Another problem is caused by storage virtualization. The "disk", as seen by the file system may in fact be a share on a SAN. In such environments the "disk" may change dynamically and in some cases, while the file system is still online. Moreover, being part of the operating system, file systems are required to be general and applicable to a variety of deployments. As a result, their heuristics need to be simple and inherently best-effort. For instance, improved interoperability with sophisticated storage appliances cannot come at the price of support for ordinary desktop environments. Maintaining consistent state on disk despite power failures, tolerating fail-stop or transient hardware faults, and adhering to expected semantics (e.g. POSIX) under edge conditions continue to be significant challenges even for simple, stable, and heavily tested file systems [4, 43, 26, 15]. As a result, most existing commercial file systems choose to exploit only coarse-grained information about the underlying hardware, like RAID stride and stripe sizes [10].

Faibish et al. suggested that varying Service Level Objectives (SLOs) could be met if the file system separated metadata into a separate region in the block address space, allowing different quality of disks to be used for each [11]. Interestingly, they suggest that an alternative,

but less desirable approach would have been to use a block level indirection mechanism which would separate metadata from data. They decided against this because it would require information about the file system semantics as well as the underlying hardware to be placed in the block layer, violating the separation between the layers. In our approach, we are able to provide this block-level indirection without requiring significant knowledge of the file system by taking advantage of in-band and out-of-band hints from the file system, while the remapping layer provides a generic mechanism which can be leveraged through policies to achieve the SLOs they identified and many other goals easily.

DualFS is a modified version of ext2 which implements metadata vs. data separation explicitly as above [24]. The metadata region is log structured, which provides journaling for ext2 and avoids an extra copy from the journal to the original location in the base file system that ext3's journal checkpointing has to provide.

Recent commercial endeavours like ZFS take a more extreme approach by providing the file system an unobscured view, and direct control of the hardware [21]. This allows it to provide powerful features such as variable block sizes and pluggable block allocation policies [6, 7]. Though impressive and powerful, this approach is a complete departure from the layered storage stack and may reduce interoperability with existing infrastructure.

6.1.2 Smarter Block Layer

In the meantime, there have been attempts to bridge the information gap from the opposite perspective, suggesting that storage hardware could provide useful optimizations and features if they gained visibility into file system semantics. Sivathanu et al. demonstrated that it was possible to infer semantic information about FFS-like file systems at the disk, which could be used to implement a variety of features including: track-aligned extents, a smarter cache for disks, secure deletion, and adding journaling support for older file systems [33]. D-GRAID uses file system semantic knowledge embedded in the storage array to place blocks in a fault-isolated manner, such that data could remain accessible despite additional disk failures than

tolerable by the RAID configuration [32].

We feel such approaches are promising, and our solution leans towards this model. However, transparently inferring file system semantics in the disk for general consumption is not trivial and requires duplicating a lot of the file system semantics in the hardware. This adds additional complications to the storage hardware, when it may not be necessary. Disk or array firmware is already complex, amounting to several thousand lines of code, and is thought to be a contributor to silent data corruption [3]. Adding file system semantics might only make matters worse. Since the file system is the most informed about its own semantics, we feel it is much more practical for it to expose some of this information to the block layer, rather than the block layer having to infer it.

I/O Shepherd pioneered the approach of piggybacking on file system journaling to implement powerful reliability and fault tolerance policies in the block layer without incurring significant performance overhead [14]. Our approach of only tagging journal checkpoint related I/O requests with hints was inspired by the Shepherd's chained-transaction mechanism. It is important to note that their approach is not truly a block-layer solution. The Shepherd was placed within the journaling layer of ext3, where it interposed on journal commits and checkpoints to implement their desired policies. In contrast, our approach simply required minor modifications to the file system so it distinguishes metadata from data, something we argue file systems should support natively. The actual policy enforcement mechanism lies entirely in the storage layer. As a result, we are able to avoid risking the stability of the file system, while accomplishing our goal of improving failure recovery times for the storage system.

6.1.3 Improved Cooperation Between File System and Block Layer

A natural middle-ground for the above contrasting approaches is a more cooperative model, where the file system exposes hints to the block layer, which the block layer tries to exploit. Our work fits in this category. However, the challenge here is to identify the right granularity at which useful information from the file system can be exposed to the block layer.

Many proposals have identified the thin block interface as a significant barrier to improved cooperation between the file system and the storage hardware. Object-based storage is the most prominent alternative [13]. These storage systems out-source layout management to the disk. The file system or other clients have the ability to specify variable sized objects and relationships between them, making informed block allocation at the disk possible. Another recent proposal was to allow file systems to suggest a range of candidate blocks as targets for writes, giving the disk freedom to choose blocks that would yield the least latency [2]. Others have shown how exposing liveness information to disks can help with secure deletion of files and improved security [31, 30].

Though the object-based interface has had some adoption in high-performance computing environments, the vast majority of disks still follow the traditional block interface. Given the momentum behind the existing interface, other arbitrary modifications to it would face far too much resistance.

However, our approach of limiting ourselves to in-band and simple out-of-band hints allows us to not require a new block interface. The in-band hints can be passed using underutilized fields in the command set. This approach is also being pursued by Mesnier et al. to integrate Solid State Disks in existing storage arrays for offering differentiated services [19].

6.2 Storage Virtualization

We used existing paradigms from the storage virtualization community to implement our system, particularly, the remapping mechanism. Block remapping has been exploited in several different contexts. Our approach is similar in design to most of these, though selectively remapping blocks based on file system hints, leveraging file system journaling for reducing online performance overheads, and the ability to execute arbitrary block placement policies, is unique to the best of our knowledge. We provide a brief survey of recent work that leverages block level remapping.

Parallax is a storage server that can scale for a large number of virtual machines [20]. It implements several interesting features like light-weight snapshotting for virtual machine disk images it hosts, as well as disconnected operation. It splits storage resources such that each virtual machine gets its own isolated block address space, and remaps requests to the corresponding physical location using an approach similar to page tables in operating systems. Our remapping mechanism's design was inspired by their work, though our need to only remap metadata blocks significantly simplified our implementation.

Other recent projects that leverage block remapping are:

BORG This is a dynamic optimizer that reorders blocks on disk based on block-level traces collected online [5].

WorkOut This is a smarter storage array that redirects incoming write requests to spare disks while a RAID array is rebuilt, thus boosting rebuild time [42].

Everest It relieves overloaded disks during peak load by redirecting incoming writes temporarily to disks with low utilization [23]. The authors used a similar approach to improve power efficiency of enterprise storage systems by allowing disks to be spun-down, and redirecting incoming write requests temporarily to active disks [22].

Our remapping mechanism could be made generic enough to subsume a lot of the features these projects provide. However, they do not provide the ability to exploit file system hints, and are not particularly configurable for general usage.

6.3 File System Consistency Check Performance

Recall that the single objective we tried to achieve in this work was reducing corruption detection and failure recovery time for existing file systems. Though most existing file systems come with a corresponding consistency checker, increases in volume sizes have left much to be desired for their performance. We have discussed the problems that the block layout on

FFS-like file systems pose for fsck and its resulting poor performance in section 2.1. Though consistency checks were largely considered obsolete with the introduction of journaling, concern about corruption in the storage stack has renewed interest in them. ChunkFS tried to address fsck performance by splitting large volumes into smaller chunks where metadata would rarely leak references across chunks. As a result, the fsck performance could be dramatically improved [16].

Recently, there has been interest in improving file system consistency check performance of existing file systems like ext3 by modifying the block allocator to cluster blocks closer together [27]. This is in many ways the ideal solution for the problem, since fast consistency verification should be a primary design goal of the file system. Though a patch of roughly 1900 lines to modify the ext3 file system had been pursued for months on the Linux Kernel Mailing List, this change was not merged despite significant support from file system developers. This highlights the resistance to modifying stable file systems which our solution side-steps.

Chapter 7

Future Work

We have demonstrated that our approach of exposing simple hints from the file system can facilitate significant improvement in failure recovery performance for existing file systems. We therefore feel confident that we can use our approach to attempt other, more lofty goals for the storage stack. We present a small selection of the further work we are considering both for the short term within the scope of this project, as well as some larger projects that may become possible due to the insights collected during this work.

7.1 Short Term

We would like to pursue more insightful block allocation strategies in the metadata region in the future. Even though we get interesting performance improvements already, we feel improved block allocation using the hints from the file system can improve performance much more. We would also like to experiment with remapping metadata blocks like inodes and study what affects they have on both online and fsck performance. We would like to conduct more performance evaluations of our system and compare against similar projects like DualFS. We also wish to thoroughly evaluate the fault tolerance and failure recovery claims we made in Chapter 3.

7.2 Longer Term

7.2.1 Integrating Solid State Disks in Existing Storage Systems

There is considerable excitement about the possibility of integrating Flash based disks in storage systems [1]. Their improved read performance and low power consumption make them interesting candidates for adoption in the enterprise. However, they also suffer from peculiar problems like limited write-endurance and the need for batched cleaning. This raises concerns about using these disks with existing file systems. File system level hints combined with our block level remapping mechanism could provide an interesting opportunity to use SSDs in new unique ways. Since seek times are not a concern with Flash, one could imagine taking periodic snapshots of metadata on existing magnetic disks and transferring them to flash to carry out quick file system consistency checks. There is also the possibility to off-load request streams to

7.2.2 Generic Block Level Remapping Mechanism

A large amount of recent research in storage virtualization has had to struggle with similar implementation challenges: providing a fast block remapping mechanism, keeping this mapping consistent despite failures, and implementing their desired remapping policy [42, 5, 22, 23]. We feel there are common primitives in all this work that if exposed through a declarative language, would allow complex storage servers to be built with relative ease.

Chapter 8

Conclusion

We have shown that simple hints from the file system, can be exploited at the block layer to meet overall goals for the storage systems, which were otherwise harder to achieve. In particular, we tried to address the classical problem of reducing failure recovery time for existing file systems, while requiring no functional modifications to file systems. Our performance evaluations indicate that our approach is able to halve the fsck time for an artificially aged file system and at the same time improve performance for a number of other metadata workloads. Our online performance evaluations based on microbenchmarks indicated that for data heavy workloads we suffered a slight performance of up to 5% on all but benchmark. We feel these results are quite encouraging and feel this approach could be useful in practise. We hope to use this technique to meet other overall goals for the storage stack, such as improving power-efficiency, performance, and reliability in the future.

Bibliography

- [1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

- [2] Ashok Anand, Sayandeep Sen, Andrew Krioukov, Florentina I. Popovici, Aditya Akella, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Suman Banerjee. Avoiding file system micromanagement with range writes. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 161–176, 2008.

- [3] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *Transactions of Storage*, 4(3):1–28, 2008.

- [4] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the effects of disk-pointer corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*, June 2008.

- [5] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. Borg: Block-reorganization for self-optimizing storage systems. In Seltzer and Wheeler [29], pages 183–196.

- [6] J. Bonwick and B. Moore. ZFS - The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.
- [7] Jim Bonwick. ZFS Block Allocation. http://blogs.sun.com/bonwick/entry/zfs_block_allocation Accessed Feb. 19, 2009.
- [8] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 217–228, New York, NY, USA, 2009. ACM.
- [9] Microsoft Corporation. Dynamic Disks and Volumes Technical Reference. <http://technet.microsoft.com/en-us/library/cc785638.aspx>.
- [10] Andreas Dilger. [RFC] Store RAID stride in superblock. <http://www.mail-archive.com/linux-ext4@vger.kernel.org/msg01774.html> Accessed Feb. 19, 2009.
- [11] Sorin Faibish, Stephen Fridella, Peter Bixby, and Uday Gupta. Storage virtualization using a block-device file system. *SIGOPS Oper. Syst. Rev.*, 42(1):119–126, 2008.
- [12] Gregory Ganger, Yale Patt, Gregory R. Ganger, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18:127–153, 2000.
- [13] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Oper. Syst. Rev.*, 32(5):92–103, 1998.
- [14] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In

- Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 293–306, 2007.
- [15] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.
- [16] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)*, 2006.
- [17] A. J. Lewis. LVM HOWTO. <http://www.tldp.org/HOWTO/LVM-HOWTO>.
- [18] Chris Mason. Compilebench. <http://oss.oracle.com/~mason/compilebench/>.
- [19] Mike Mesnier, Scott Hahn, and Brian McKean. Making the most of your SSD: A case for Differentiated Storage Services. http://www.usenix.org/events/fast09/wips_posters/mesnier_poster.pdf.
- [20] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008(Eurosys '08)*, pages 41–54, New York, NY, USA, 2008. ACM.
- [21] Sun Microsystems. Zfs. <http://opensolaris.org/os/community/zfs>.
- [22] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: practical power management for enterprise storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

- [23] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony I. T. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 15–28. USENIX Association, 2008.
- [24] Juan Piernas, Toni Cortes, and José M. García. Dualfs: a new journaling file system without meta-data duplication. In *Proceedings of the international conference on Supercomputing*, pages 137–146, 2002.
- [25] Juan Piernas and Sorin Faibish. Dualfs: A new journalling file system for linux. In *Proceedings of the Linux Storage & Filesystem Workshop*, feb 2007. <http://ditec.um.es/~piernas/dualfs/presentation-lsf07-final.pdf>.
- [26] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005.
- [27] Abhishek Rai. Re: [CALL FOR TESTING] Make Ext3 fsck way faster [2.6.24-rc6 - mmpatch]. <http://lkml.org/lkml/2008/1/23/38>.
- [28] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 259–274, Berkeley, CA, USA, 2002. USENIX Association.
- [29] Margo I. Seltzer and Richard Wheeler, editors. *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings*. USENIX, 2009.
- [30] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-safe disks. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 15–28, 2006.

- [31] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or death at block-level. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 26–26, Berkeley, CA, USA, 2004. USENIX Association.
- [32] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving storage system availability with d-graded. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 15–30, Berkeley, CA, USA, 2004. USENIX Association.
- [33] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies*, pages 73–88, 2003.
- [34] Keith A. Smith and Margo I. Seltzer. File system aging—increasing the relevance of file system benchmarks. *SIGMETRICS Perform. Eval. Rev.*, 25(1):203–213, 1997.
- [35] Gokul Soundararajan, Daniel Lupei, Saeed Ghanbari, Adrian Daniel Popescu, Jin Chen, and Cristiana Amza. Dynamic resource allocation for database servers running on virtual storage. In *FAST'09: Proceedings of the 7th conference on File and storage technologies*, pages 71–84, Berkeley, CA, USA, 2009. USENIX Association.
- [36] Lex Stein. Stupid file systems are better. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.
- [37] N. Talagala, R. Arpaci-Dusseau, and D. Patterson. Micro-benchmark based extraction of local and global disk characteristics. Technical report, Berkeley, CA, USA, 2000.
- [38] Stephen C. Tweedie. Linuxexpo '98 journaling the ext2fs filesystem page 1 journaling the linux ext2fs filesystem, 1998.

- [39] Wouter Verhelst. Network block device. <http://nbd.sourceforge.net/>.
- [40] Ric Wheeler. Re: [CALL FOR TESTING] Make Ext3 fsck way faster [2.6.24-rc6 - mmpatch], January 2008. <http://lkml.indiana.edu/hypertext/patches/kernel/0801.1/3174.html>.
- [41] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of scsi disk drive parameters. *SIGMETRICS Perform. Eval. Rev.*, 23(1):146–156, 1995.
- [42] Suzhen Wu, Hong Jiang, Dan Feng, Lei Tian, and Bo Mao. Workout: I/o workload outsourcing for boosting raid reconstruction performance. In Seltzer and Wheeler [29], pages 239–252.
- [43] Junfeng Yang, Can Sar, and Dawson Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, 2006.