DESIGN AND IMPLEMENTATION OF THE *CHUNKS* FEATURE

by

NAWAR A. NORY

B.Sc., Al-Mansour University, Baghdad, 2005

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Approved by:

Major Professor
Daniel Andresen

Copyright

Abstract

The recovery-driven design of the filesystem has been one of the most challenging fields over the major trends in operating systems. This field has assumed considerable importance in the past decades as the disk sizes have been increasing without a comparable increase in the disk I/O bandwidth and seek time. The rapid increase in the storage size is expected to become constant in the future due to the growing market demand and the continuous database size increment of many companies and major businesses. Due to the same reason, the cost of the average filesystem checking time has increased without a significant improvement in the disk I/O bandwidth and seek time performance. Operating system bugs, power outages, and hardware failures which result in a filesystem crash were the main reasons behind the innovation of novel recovery approaches such as *Journaling* and *soft-updates*. Although such approaches avoided complete filesystem checking by checking solely inconsistentices in filesystem metadata, it become inevitable for them to check the entire filesystem for inconsistencies because of the previously mentioned types of problems. One of the emerging recovery-driven designs which considers minimizing filesystem checking cost is the *Chunkfs* filesystem. *Chunkfs* filesystem introduces an innovative look into the filesystem design by dividing the filesystem layout into smaller chunks, each one of which represents a smaller scale filesystem by itself.

In our work we probed an alternative recovery-driven design which is considerably inspired by the *Chunkfs* concepts and follows the same design guidelines. This recovery-driven design is introduced by adding a new feature to the filesystem which best utilizes the existing underlying design through considering the blockgroups as individual chunks, confining their files and directories spanning across different blockgroups by means of special controlled continuation links. These links provide a fault isolation means by circumscribing the

checking of the filesystem to only these blockgroups which appear to be dirty after a crash, a method resulting in a moderate reduction in filesystem checking cost. We also probed different metrics of metadata sizes, and the probable cost of files and directories expansion across the different blockgroups.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere thanks to my major professor, Dr. Daniel Andresen, who I always considered a great mentor and guide through my Master's degree, not only by my testimony, but by the testimony of others. I appreciate his generosity with his time, and I thank him for providing me with the department resources to help alleviate the difficulties of my thesis work.

I would also like to show my sincere appreciation and thanks to my committee members, Dr. Gurdip Singh and Dr. Mitchell Neilsen, for their constant help and patience during the course of my Master's study.

Finally I would like to thank all those great faculty members, fellow graduate students and friends of K-State for their understanding, support, and respect along the line.

Dedication

I would like to express my sincere thanks to Buthaina and Abdul-Ilah for their stamina in providing me and my brothers with the best upbringing and for being the loving parents who always showed their careful nurturing and taught me the most valuable lessons of life. To you this work is dedicated.

Finally I want to thank my brothers, Bassam and Bashar, for all the support, laughter and mirth, especially my big brother Bassam, who I sincerely thank for allowing me to use a partition on his personal computer to install and use my first Linux. Last but not least, I would like to thank my extended family and all those people who formed the shape of my life.

Chapter 1

Introduction

The content of this chapter is intended to be an introduction to the filesystem, its features and recovery techniques. It also introduces the motivation behind this dissertation and the various aspects of the work.

1.1 Data And Filesystem Trends

Filesystem is the crucial part of any operating system, is the main concern in the system since it stores the various kind of information. This data is usually stored as *blocks*, more precisely, *data blocks*. The *data blocks* form the *file*. The availability of *data blocks* is a prerequisite to storing the corresponding *file*. Each file is described by a legacy data structure known as *inode*. The *inode* usually exists on disk. Then the content of the *inode* data structure is copied to the memory when reading, writing, or truncating the file since the likelihood of requiring the specific inode in the future is high. The basic functions of the filesystem are to store the files, preserve the filesystem integrity, and recover the content of the files if the filesystem crashes due to an operating system bug, power failure or other reasons. These are the main guidelines that any file system designer should adhere to.

The design of the filesystem should conform to the common rules of filesystem design; that is, each filesystem should provide consistent representation for the data stored on the disk with which the filesystem recovery process should be familiar such that the data can be rolled back to a previous consistent state.

As mentioned, the basic scenario that leads to an inconsistent state of a filesystem is the crashing of operating system due to either a power-failure or a malfunction (operating system bug). However, the operating system should provide the required means through which it can recover the filesystem to a recent consistent state. This is exemplified through a filesystem-checker. The filesystem checker checks all the metadata information which composes the filesystem and checks if there are any inconsistencies such as incomplete writing of the inode hard link counter. When creating a hard link to a file, there are two possible scenarios, in the first, the hard link counter of the file inode is increased first and then the failure happens before the hard link is added to the list of entries of the containing directory. In this case the directory is consistent. On the other hand, if the scenario is the reverse, we end up with an inconsistent state, in the sense that deleting the hardlink removes all the related data blocks of the specified inode without updating the directory entry list (i.e., there will be a directory entry which refers to the non-existent hardlink). Using the inode number of the deleted hardlink to create another file may lead to unexpected corruption to the content of the newly created file.

Numerous approaches were developed to provide a seamless and consistent recovery for the filesystem metadata. Most of these were provided to minimize the time required to check and recover the filesystem integrity using filesystem integrity checkers. Such approaches are presented in [2, 3] .

One of the reasons behind the constant motivations to improve filesystem recovery such that filesystem checking is neither expensive nor time-consuming is the constant increase in disk-size. According to Kryder Law [1] , the capacity of the magnetic hard disk doubles annually. As illustrated in Table 1.1, the hard disk sizes are increasing in large magnitudes compared to transfer rate and seek time. This illustrates the fact that the harddisk transfer rate is not increasing proportionally compared to both the disk size and the seek time which are increasing relatively slowly. This gives a clue to the fact that as a result the filesystem checking time will increase rapidly as the filesystem checking time is proportional to the

	2005	2009	2013
Drive Capacity (GB)	500	2,000	8,000
Transfer Rate (Mb/sec)	995	2,000	5,000
Read Seek Time (ms)	8	7.2	6.5

Table 1.1: *Hard Disc Drive Trends [1]*

size of the metadata on-disk and to the disk size. This also refers to the upcoming increase in downtimes due to the increased filesystem checking time during the offline mode of the filesystem.

1.2 Thesis Objective

The main objective for this thesis is to reduce the filesystem checking time. This goal is achieved by implementing a new feature in the filesystem design which reduces the checking-time of the filesystem data and metadata integrity.

1.3 Motivation

The motivation behind the thesis is to provide an alternative design for the *Chunkfs* filesystem by implementing a *chunkifying* feature for the filesystem. This is achieved through the usuage of the underlying functionality and the design of the filesystem rather than introducing a different design for the filesystem. The basic two design techniques which prevent the time-consuming task of filesystem checking are either minimizing the checking time or preventing file system checking in the first place using approaches like *soft-update* [4] or *journaling* [2] . Online checking approaches [5] have also been developed. However, these techniques depend on continuous updates to the metadata on disk which in some cases become expensive. Despite the fact that the provided approaches strive to eliminate filesystem checking, filesystem checking becomes a necessity in some cases of failure. Such failures increase the operating system downtime due to the prerequisite of the filesystem to be in offline mode when it needs to be checked. Thus the work of *chunkfs* recovery-driven design

was introduced as a candidate solution to minimize the checking-time as a best resort.

1.4 Approach and Previous Work

The eventual goal of this work is to minimize the cost of the comprehensive checking of the filesystem integrity in an efficient manner. This is achieved by dividing the filesystem into fault-isolated block groups with controlled continuation links to other blockgroups. Thus, we chose the name *chunks* to refer to the fault-isolation nature of these block groups with their modified behavior. This has the benefit of having a smooth filesystem checking, as the checking domain will be confined to only these blockgroups whose metadata or data was modified during half-updates through the time the operating system was crashed.

Due to the tendency of regular files and directores to grow as a result of varying write and update operations, the regular files and directories should be allowed to grow seamlessly. Such increase in file size requires their spanning across other chunks in order to allocate some free space owned by these chunks. In order to allow such spanning, a new type of conrolled continuation links is introduced, this type of links is referred to as *Continuation Inodes* or shortly *cnodes*. The *cnodes* are extra inodes located in these chunks and assigned to the original file's inode. They utilize the free space which exists in their owner chunks and allow for that space to be counted towards the full size occupied by the file. This has the penalty of inode usuage increase over the entire filesystem, and probable disk seek time increase as the disk heads are going to move back and forth among the cylinder groups to read the different inodes.

This work is considered a continuation to the recovery-driven design of the experimental *Chunkfs* file system [3] and strives to complement it. Moreover, it conforms to the same goals which motivated the *Chunkfs* filesystem design. These goals are:

- Provides a new novel bird's-eye view of filesystem design whose main goal is to ensure a seamless filesystem recovery.

- Understands the ins and outs of filesystem designs in order to bring the most efficient way to apply the recovery-driven technique to them.
- Understands the penalties which accompany such design, such as major changes in the filesystem metadata components and their effects on the general filesystem functionality.

Chapter 2

File System Architecture

This chapter provides an overview of the file system generic architecture. This architecture is general for most UNIX variants despite the slight difference in the name conventions which are used in some of these Unixes. These name conventions are the technical terms which are used to describe the design components that form the file system structure. These technical terms became so common among the Unixes such that they become the de facto terminologies used in the design and the descriptions of most of the Unix filesystems. Thus we are going to explain these terms and the design decisions which stands behind them.

The filesystem architecture consists of multiple important components (structures). These components describe the filesystem design and exist on-disk in the offline mode, whereas in the on-line mode, some of these components which are accessed frequently are copied to the memory. Thus, these components and the operations they perform have an effect on the function of the filesystem in general.

The filesystem layout is partitioned into a number of groups. Each one of these groups refer to a specific number of blocks to avoid filesystem fragmentation as the filesystem strive to keep the data blocks which belong to a specific file in the same block group of that file. Each block group consist of the information below:

- The filesystem's *superblock* copy
- filesystem's *blockgroups* group copy

- `data block bitmap`
- `inode bitmap`
- `inode table`
- `data blocks`

The following subsections provide a brief description of the main components of the filesystem[6] .

2.1 The Superblock Structure

The *superblock* structure describe a specific mounting point of the filesystem and stores most of the important information which is related to it. The *superblock* is located on a special sector on the disk which follows another important sector which stores the *bootblock*. The *superblock* is replicated over the blockgroups which comprise the filesystem in order to provide a backup for the primary superblock in case of being damaged due to filesystem crash or failure. The superblock usually contains the following fields:

- `filesystem block size` – and its corresponding size in bits.
- `superblock operations` – the operations which are related to superblock (much like member functions in the Object-oriented paradigm)
- `magic number` – filesystem magic (identification) number
- `list of used inodes`
- `pointer to other superblocks`
- `mount point of the filesystem`
- `number of blocks per group and block`

- `number of inodes per group and block`
- `number of block groups in the filesystem`
- `filesystem state` – determines whether the filesystem state is dirty or not
- `last checked-time` – which determines the last time the filesystem was checked for inconsistency. This field is among the fields which are checked at the boot time as it may determine whether the filesystem needs to be checked for inconsistency after a specified number of mount points or after a specific time interval.

2.2 The Inode Structure

The inode is the most important data structure in the filesystem, as each inode corresponds to a particular file in the filesystem, and the inode is unique such that two files can not share the same inode. The inode structure store important information about the file. This information spans along the lifetime of the file which is also the case for the inode itself. The important information stored in the inode structure about the file which it addresses is as follows [7] :

- `inode number` – a unique number which identifies the inode and the file associated.
- `file size` – size of the file of this inode.
- `time-based fields` – describe the time of the last access and write to the file and the time of modification to the inode.
- `link count` – number of links which refer to this inode's file.
- `inode operations` – describes the operations which can be performed by the inode object, which are similar to the member functions in the Object-oriented paradigm.
- `file operations` – these are the operations which are responsible for file manipulation.

2.3 The Directory Entry

One of most important structures in the design of the filesystem is the directory entry or shortly dentry. This structure is a special type of file (e.g regular file) which stores a list of files. Its representation is used to describe the components of a pathname, as each component is represented as a directory entry. Thus the filesystem directory hierarchy is mainly represented as a root directory "/" which contains a list that represents the complete directory tree of the filesystem. Each directory entry structure stores the following information.

- `directory entry inode` – inode associated with this dentry.
- `list of unused dentries`
- `counter` – usage counter for the dentry object.
- `reference to the parent of this dentry`
- `list of the subdirectories (children dentry objects)`
- `operations associated with the dentry object.`

Each directory entry has four states: *free*, *unused*, *in use*, and *negative*. Moreover, some of the most important directory entries are the "." and the "..". The first one always refers to the parent dentry which contains the current directory as entry, and the other is used as a reference to the current directory entry. For the root directory the "." directory entry refers to the root itself.

In addition to the above data structures there are other data structures which pertain to the filesystem such as *dentry cache* and *file* data structures. There are different data structures which are used in block indexing, which differs according to the filesystem, for an example, the *B+ tree* in reiserfs [8–10] and *bitmaps* [11, 12] in both the ext2 and the ext3 filesystems.

In the following section we will take a look at the ext2 filesystem hierarchy and examine its structure in detail; we will also mention the filesystem checker of the ext2 (e2fsck) [13] and examine its steps.

2.4 Second Extended File System (ext2)

The second extended files system (ext2) [11] is one of the first filesystem which was adopted for Linux; thus it is native to it. Ext2 filesystem was intended to reslove some of the demanding challenges which faced the Extended filesystem (ext), to which the ext2 is an extension. Ext2 has proved its relability, as it is has been used since its inception for different kinds of production machines. The feature presented in this thesis is implemented toward the ext2 filesystem.

2.4.1 Ext2 Architecture

Ext2 filesystem layout was inspired by previous implementations, mostly the Fast Filesystem(FFS) [14] . The disk layout, most of the data structure and concepts of the ext2 were exported from FFS design. Although the ext2 filesystem has gone through constant changes in the disk layout, the data structures have left unchanged.

The disk layout of the ext2 filesystem is shown below in Fig. 2.1. As it is clear from the figure, each blockgroup contains a copy of the superblock, which is used as a backup for the primary one and a copy of the group descriptors of the filesystem. It also contains the block bitmap, the inode bitmap, the inode table and the data blocks.

Each bitmap is used to provide a mapping for the inodes in the inode table and the blocks in the data blocks, that is, each entry (bit) in the bitmap is either 0 or 1, where 0 refers to a free block or inode, whereas 1 means that it is occupied.

Moreover, the size of the block in the filesystem determines the size of the bitmap, which always occupies a one block [1]. It also determines the total number of both blocks and inodes

[1]The default block size is 1024-bytes, whereas the maximum supported size is 4096-bytes

which exist in a blockgroup and the filesystem in total; i.e., a block size of 2048 refers to the existence of 16384 blocks-size blockgroups in the filesystem [6] .

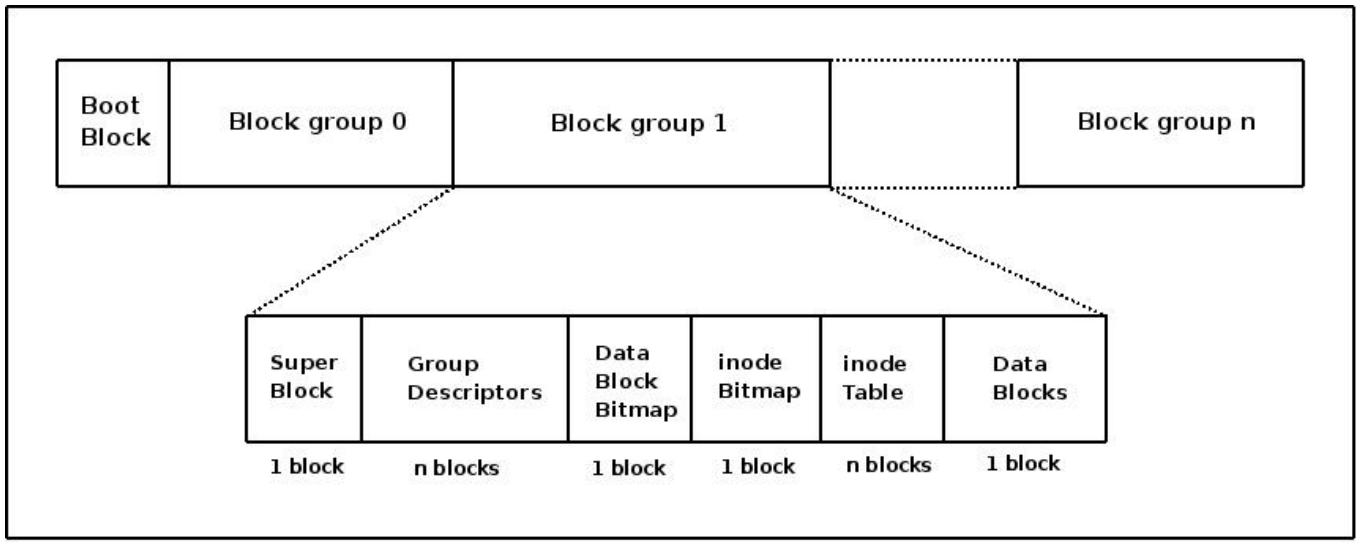

Figure 2.1: *Ext2 Disk Layout [6]*

2.4.2 Basic Disk Managing Operations

One of the most important consideration when it comes to designing a file system is the *file fragmentation* prevention. *File fragmentation* happens when files are fragmented across the filesystem in non-adjacent blocks. This increases the I/O latency, as the disk head must move back and forth among different positions on the disk. Thus the designers of the ext2 filesystem took this factor into consideration when implementing the basic file creation, modification and expansion operations in order to achieve a fair implementation.

File & Directory Creation

Since the directories are a special kind of files, both file and directory share the same creation operation. The creation procedure strives to locate the new file's inode in the same blockgroup where the parent directory of the file is located. The creation procedure balances the number of regular files and directories in a specific group according to a specific criterion.

If we want to create a new home directory for the user "nawar" in the "/home" directory, i.e., obtaining the pathname "/home/nawar," the creation procedure is as follows:

- Since "/home" is not the root directory, the creation procedure strives to locate the "nawar" directory in the same blockgroup of the "/home" according to the following criteria: the number of folders did not exceed a predefined threshold, the number of free inodes is sufficient, and the number of regular files and directories in the blockgroup is balanced.
- Search for the first free *ith* bit in the inode bitmap of the blockgroup in question.
- Allocate the corresponding inode for the file and mark the corresponding free bit of the inode bitmap found in the previous step as used and mark the inode bitmap as dirty.
- Decrease the number of used inodes in both the containing blockgroup and the superblock of the filesystem and increase the number of used directories in the containing blockgroup.
- Mark the the containing blockgroup and the superblock as dirty.
- Create the directory entry "nawar" and insert into the list of directory entries owned by "/home."

File & Directory Deletion

As one can imagine, the file and directory deletion is the reverse of the creation operation. However, this step is done as the finall pass of a series of clean-up operations which mainly free the data blocks assigned to the specified file or directory and truncate the file. These steps are followed:

- Remove any dirty buffers associated with the data blocks of the file.

- Compute the index of the blockgroup containing the file in order to access the inode bitmap.
- Increase the number of free inodes in the containing blockgroup and decrease the number of used directories
- Clear the bit corresponding to the file's inode in the inode bitmap.

2.5 Disk Recovery

Filesystem do not perform flawlessly in most cases. Many filesystem errors may occur during the course of operation which can immensely affect its performance and lead to filesystem crash in the worst case. Other factors which cause the crash of operating systems are power outage, hardware errors or other bugs in the operating system.

Such problems were taken into consideration when designing filesystems. Thus the goal for operating system designers is to implement an efficient error-check tool which can probe the integrity of the filesystem and recover it to a previous consistent state. However, filesystem checking may not be desirable for a large-volume filesystem, as the time required to recover it may span many hours or even days. Many approaches were introduced to prevent filesystem checking in the first place, however, in some cases filesystem checking becomes inevitable.

2.5.1 Fsck

Fsck is the most recognizable filesystem repair and recovery program. This program uses the redundant structure information in the UNIX filesystem to perform an exhaustive check for the filesystem [15] . If during the check process, inconsistencies are discovered, the filesystem *fsck* interactively prompts the user to choose whether to fix the encountered inconsistency or to pursue the check process. This procedure is repeated until the filesystem is repaired according to the user request and the assumption made by checker program.

The checkers implemented for BSD [15] and EXT2/EXT3 filesystem [13] follows the same organization of the checker implementation, which is composed of five passes where the early passes check and provide information that is required for the later ones. The *fsck* passes are explained below.

Pass 1

In Pass 1, most of the information which is required for the consecutive passes of *fsck* is collected here. The main information collected here is related to the inodes in the filesystem. These inodes are collected in a list and checked for the consistency of its mode, size and the associated data block.

Moreover, during this pass the blocks of the inodes are checked if they are used by other inodes, *i.e.*, duplicated blocks which are claimed by other inodes. In this case the solution is either duplicating the shared blocks or releasing either of the sharing inodes. The different information about the inodes and the blocks in the filesystem is collected in different bitmaps during this pass in order to be used in further passes to avoid the I/O operation related to inode information fetching. This bitmap information, is used during the course of checking to ensure its consistency with the on-disk bitmap information.

Pass 2

In this pass, the bitmap information collected regarding the directory inodes in pass 1 is used here to traverse the list of directory inodes in the filesystem. During this traversal, a number of sanity checks are applied to the directory inodes:

- The length of the directory entries should be within legal bounds.
- The length of the directory name should be withen legal bounds.
- Using the information collected from pass 1, the directory inode number should be flagged as in-use inode and its number should be within legal bounds.

- Check for the existence of both "." and ".." directory entries and ensure their consistency. More information about links counts is collected in this pass in order to be checked in a later pass.

Pass 3

In this pass, directories' connectivity to the root directory "/" is probed. During the course of this pass, the tool verifies if the directory is disconnected and if it is causing a filesystem loop. In both cases, and in order to break the filesystem loop in the latter case, e2fsck will disconnect these directories and attach them to the *lost+found* directory.

Pass 4

In this pass, inode link information obtained in pass 2 is probed. Each piece of collected inode link information is paired to the actual inode link information on the disk. Inodes with incorrect link count or null link count field are corrected here. Inodes with null link count are disconnected from the tree and attached to the *lost+found* directory.

Pass 5

In this pass, the collected inodes and block bitmaps are checked for consistency with the ones that exist on-disk. If any inconsistency is encountered, the on-disk bitmaps are corrected accordingly.

2.5.2 Other Recovery Techniques

As mentioned previously, some approaches were developed to prevent the *fsck time crunch* [3] for the filesystem. Some of the most general and widely-used approaches are presented here.

2.5.3 File System Journaling

To avoid the exhaustive and time-consuming consistency check by fsck which is required for large-volumes disks, a new design for filesystem was required which handles failures and recovers filesystems in a timely manner. This design is known as filesystem journaling.

The journal is an area in the filesystem where recent disk writes are logged. Filesystem journal is useful to recover the filesystem from failures during write operations which occur very frequently. The common scenario when a crash occurs is that the data which was about to be written to the disk is stored on the journal (write operation is called *transaction*) so that when the crash occurs the filesystem is able to return the information which was stored on the journal back to the filesystem (replaying the transaction). To be precise, the data committing operations are performed in two steps:

- The data blocks of a file to which the write operation is performed are stored in the journal.
- When the previous operation finishes, the blocks are written back to the filesystem. When the latter finishes, the data blocks are removed from the journal.

There are two types of data which can be stored in the journal, these are normal data and metadata. As mentioned previously, the metadata is the one which is related to the file status. According to the kind of data to be logged, there are three journalings modes, as there is a trade-off between the speed of the logging operations and the type of data to be logged. The three types of journaling modes are:

Journal Mode This type of journaling is the slowest and the most expensive since both the data and the metadata are logged to the journal. Since both types of data are logged, this comes at the expense of the number of disk accesses and hence on the speed of the logging operation which is considered the slowest.

Ordered Mode This is the default journaling mode as the data is written to the disk before the metadata is marked as logged to the journal. This mode depends on the assumption that most files are appended rather than overwritten; hence, under this assumption the data which is written to the file is not committed to the journal during the crash and is not considered during journal replay. Thus, the file is rolled-back to

its previous consistent state. However, in the second case when the file is overwritten, disk corruption may significantly occur, as the file will be in an inconsistent state.

Writback Mode This is the mode which is considered the fastest yet the most critical, as only the metadata are logged to the journal. As noticed, the logging operation is the opposite of the previous one, however, it is the most critical. For example, if there is a crash after metadata has been committed to the journal and before the data has been written to the disk, the metadata will reflect inconsistent information and we will end up with a corrupted file.

The typical scenario when the file system becomes corrupted is that there is unclean mounting during incomplete updates for data and/or metadata. In the next boot time the filesystem's journal is probed for any inconsistency in the filesystem data and/or metadata (based on the previous three modes). Accordingly, it is decided whether the half updates and transactions contained in the journal should be replayed or not. In the worst case scenario, if an utter corruption has occurred which is unrecoverable by the journal, it becomes unavoidable to issue a comprehensive diagnosis and check the entire filesystem in order to detect and repair any massively corrupted data or metadata.

2.5.4 Soft Updates

One of the important approaches in filesystem recovery, which avoids write-ahead logging and journaling is the *soft-updates* approach. The soft-updates is an asynchronous write-back approach which is proposed as a replacement to other techniques, among them its counterpart synchronous write-back approach. The operation of the synchronous write consist of writing back dirty metadata in cycles called *write-back cycles* in order to maintain a coherent metadata in both disk and memory [16] . However, the synchronous writes approach has always been questioned since the writes proceed at disk speed, which is indeed much slower than memory and CPU speeds, imposing a serious performance penalty [4] .

In order to overcome such weak performance, a smarter approach, exemplified by soft-updates, is introduced. For soft-updates, the filesystem uses delayed metadata updates, tracks the dependencies associated with these updates and enforces these dependencies during writebacks as required. Since the likelihood of cyclic-dependency existence is relatively high on filesystem block level, soft-updates move around this problem by tracking dependency on pointers-level in memory.

As with each approach there are inconsistencies, the soft-updates is not an exception. The main sort of inconsistency which may occur because of the soft-updates mechanism is the unreferenced blocks and incorrect inode links counts; nevertheless, the filesystem can operate normally. The suggested solution for overcoming such inconsistency is to run a special version of **fsck** in the background [17] , which takes a snapshot of an online-filesystem, probes it for unclaimed blocks and inodes with incorrect links counts, preforms the necessary repairs, and releases the snapshot. Despite the soft-update approach, problems like hardware and filesystem bugs necessitate performing a comprehensive filesystem check in its offline mode.

2.5.5 Non-Volatile RAM

The non-volatile nature of the NVRAM allows the maintenance of the on-disk data during power outage, as it acts as a continuously powered RAM. This alleviates the process of writing back metadata information constantly to disk in synchronous or asynchronous manner, as it is enough to maintain the consistency of data updated on the NVRAM [18] , with the update operation running in any desired mode.

One of the most important penalties that comes with NVRAM is that crucial metadata stored on it is subject to corruption in case of NVRAM failure. In that case, it is advisable to perform a complete comprehensive integrity check on the filesystem, which is subject to the size of both the data and the metadata.

2.5.6 Others

Some other approaches were introduce to maintin the consistency of metadata in the most efficient manner. Among these techniques is "the episode filesystem" [19] . Others include shadow paging [20] and reducing fsck time [21] . These approaches tend to avoid performing a comprehensive filesystem integrity check; however, it becomes mandatory in cases like media-failure and operating systems bugs.

Chapter 3

The Design Of File System Chunks Feature

In this chapter we are going to introduce the recovery-driven design which was inspired by *chunkfs* [3] filesystem design. The paradigms introduced here differ significantly from those introduced with the design of *Chunkfs*; however, we used the same name conventions which were coined for *Chunkfs*.

3.1 Recovery-Driven Design

The main idea of the *Chunkfs* filesystem is to divide the filesystem into separate chunks each of which represents a smaller instance of the filesystem itself. The main motivation behind this work, which was inspired by the previous idea, is to provide an alternative *chunking* technique through which the likelihood of heavily altering the structure of the file system is not required. Thus the idea is to find a more efficient technique, which utilizes the underlying structure of the filesystem and reflects the goals of *Chunkfs*

The design implemented here utilizes the idea of *chunks* which already exists in the filesystem, that is, the filesystem itself is divided into separate *blockgroups*. Each of these blockgroups can be considered as a chunk by itself. Since the idea behind *chunkifying* the filesystem is to provide fault-isolation boundaries such that the filesystem checker skips those *chunks* which were not active during the crash time and check the consistency of only those

chunks which were active. The new design approves this through confining the boundaries of these *chunks* or *blockgroups* such that the rules to make the links among the blockgroups are quite obvious.

As is showing in Fig. 3.1 and with minimum change to the ext2 filesystem structure view, each *blockgroup* is represented as an independent *chunk*, and the whole filesystem is represented as a container of these *chunks*. The idea of fault-isolation of the *blockgroups* is implemented through limiting the linkage among these *blockgroups*. Allowing such limited linkages enables the checker to easily correct the *blockgroups* of the corrupted metadata by tracking only its linkage to other *blockgroups*; thus, this shall provide a faster access to the *blockgroups* which are relative to the corrupted ones without the need to traverse all the blockgroups of the filesystem searching for them.

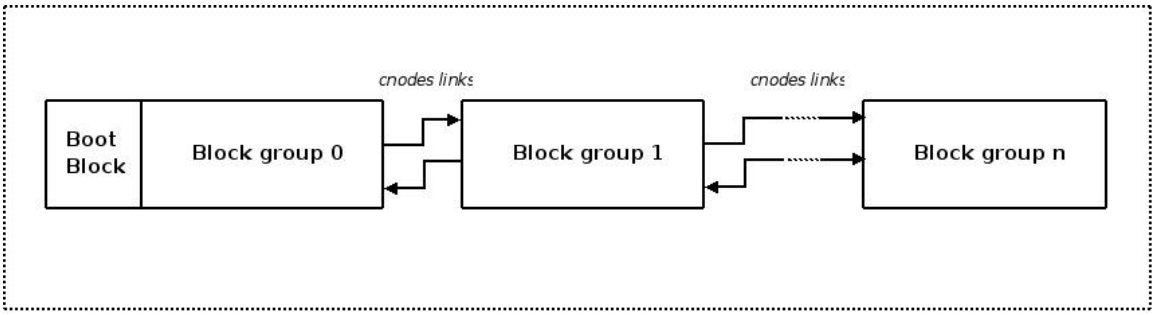

Figure 3.1: *Layout of the Ext2 partition*

3.2 Continuation inodes

In order to put the previous design into action –that is, the splitting of the filesystem into multiple chunks with fault isolation boundaries– a question may arise: when a regular file or directory needs to grow, how the filesystem will handle that?

The answer to this question is exemplified through the usage of *continuation inodes* or *cnodes*. The typical condition which requires the spawning of a *cnode* is when a regular file needs to expand due to write or truncate operation, and when there are not enough free data blocks in the current blockgroup (chunk). The kernel bounds to search the remaining

blockgroups of the filesystem starting from the current one for both free blocks and free inodes. Once a free inode and free blocks are located in a blockgroup, the kernel allocates an inode and connect it to our original inode. Thus, the newly allocated inode is called *cnode* and the original inode which is linked to it is called *parent* or *parent* inode.

These two types of inodes are similar to the ordinary inodes, however, they are differentiated by the type of the flags they are marked with, that is there are two types of flags: EXT2_HAS_CNODE_FL which denotes the *parent* inode and EXT2_CNODE_FL which denotes the *cnode*.

As Fig. 3.2 illustrates, the *cnodes* are repsresented as a list data structure where each *cnode* in that structure has a forward and backward pointer. The forward pointer of the last *cnode* points to the *parent* inode and the back pointer of the *parent* inode points to the last *cnode*. Each of these *cnodes* exists in either the same blockgroup or a different one according to the availability of both unused inodes and blocks, *i.e.* they are located in $group_i$ where $i \leq group\_desc\_count$, such that $group\_desc\_count$ denotes the number of blockgroup descriptors which provide information about the filesystem blockgroups.

The reason behind using such structure is to minimize the spread of datablocks which belong to a specific file over the filesystem blockgroups, thus minimizing the filesystem fragmentation. This is controlled through the linked-list data structure in the sense that it is much easier to traverse this list which belongs to a specifice *parent* inode rather than traversing all the blockgroups of the filesystem. In terms of running time, the running time of filesystem blockgroup traversal is $\Theta(group\_desc\_count)$ in both the worst and the best case. This running time is governed by the number of blockgroups which is significantly larger than the number of *cnodes* attached to a particular *parint* inode[1].

[1]For the purpose of testing, the design decision was to limit the number of *cnodes* to 4, which is, of course, less than the number of blockgroups of any filesystem on any production machine.

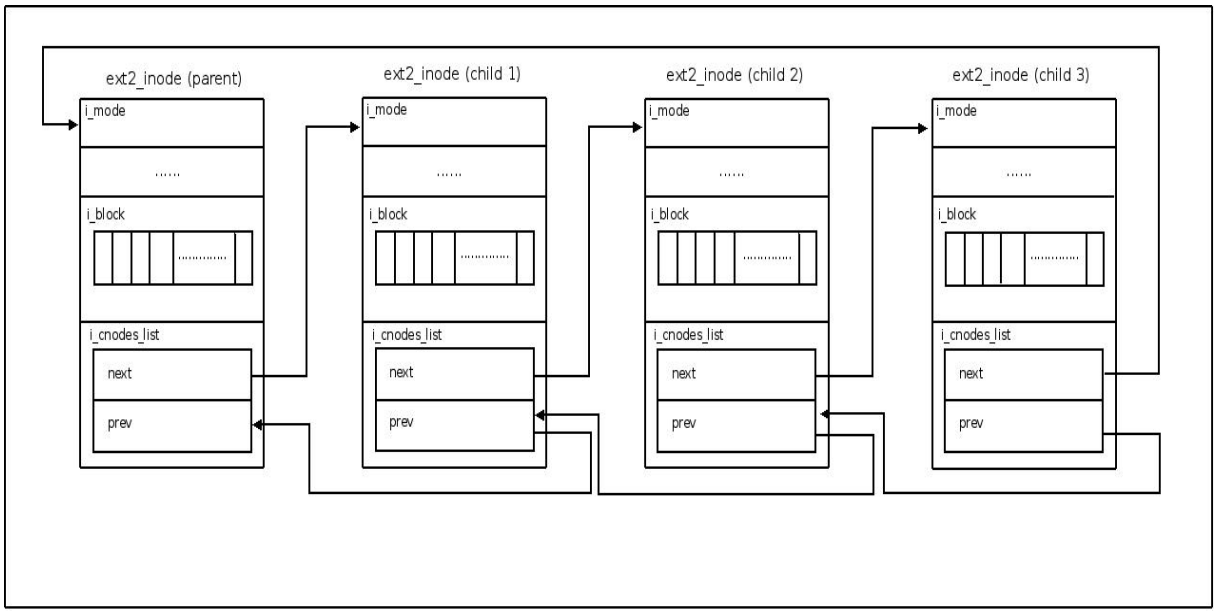

Figure 3.2: *Continuation inode list*

3.2.1 Handling Regular Files Expansion

The main reason behind the creation of *cnodes* is the need for a file to expand due to the lack of data blocks in its local blockgroup. Consider a scenario where we have two inodes, $inode_i$ and $inode_j$, such that $i \neq j$ and $inode_i$ belongs to $group_i$ and $inode_j$ belongs to $group_j$. When a file which is represented by $inode_i$ needs to expand due to a write operation, and when the kernel can not locate free blocks in $group_i$, it allocates $inode_j$ in $group_j$ where there are free data blocks and connect it to $inode_i$ which will be its *parent* inode. Then the free data blocks which belong to $inode_j$ are counted towards the total data blocks owned by $inode_i$ and as illustrated in Fig. 3.3.

3.2.2 Handling Directories Expansion

The directory is a special kind of file which has a list of other files or directories. When a new file is being created, and there are no enough free data blocks in the current blockgroup to store the directory entry which refers to the new file, a cnode is created in another blockgroup which has enough free data blocks and inodes. In a similar scenario, we have

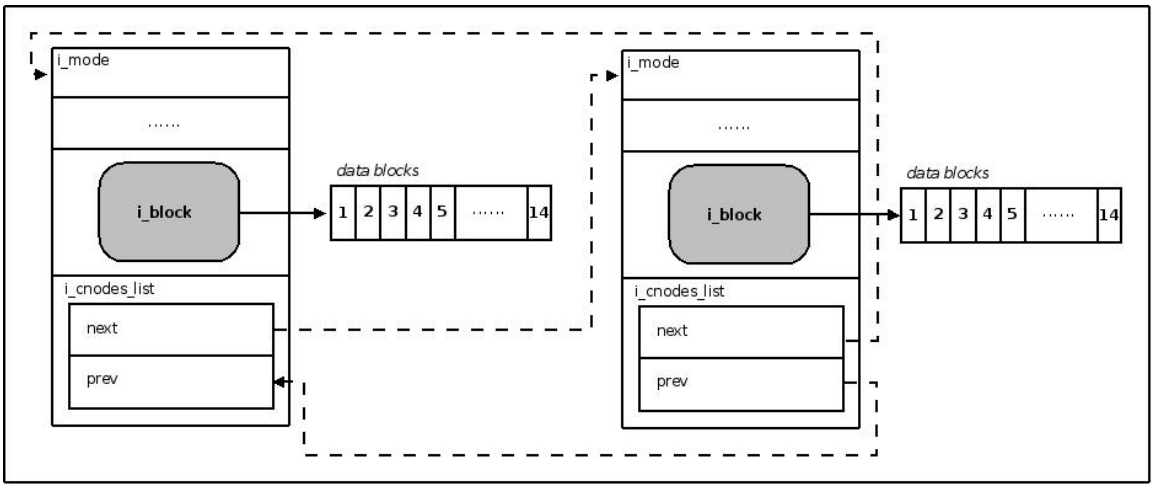

Figure 3.3: *The cnodes spanning*

two inodes, a directory $inode_i$ and $inode_j$, such that $i \neq j$ and $inode_i$ belongs to $group_i$ and $inode_j$ belongs to $group_j$. When a directory which is represented by $inode_i$ needs to expand due to insufficient space to store the directory entry of the new file, and when the kernel can not locate free blocks in $group_i$, it allocates $inode_j$ in $group_j$ where there are free data blocks and connect it to $inode_i$ which will be its *parent* inode. Then the free data blocks which belong to $inode_j$ are counted towards the total data blocks owned by $inode_i$, as also referred to in Fig. 3.3.

3.3 Chunkfs Feature Implementation

In filesystem development, the inclusion of a new feature in the filesystem requires developing the support for that feature in three main parts or phases of the filesystem (Fig. 3.4):

- *mkefs*: Filesystem-creation-and-making-tool.
- *Filesystem Driver*: which is the driver that makes the filesystem available for usuage (mostly included in the kernel filesystem tree).
- *fsck*: Filesystem checking, reparation, and recovery tool.

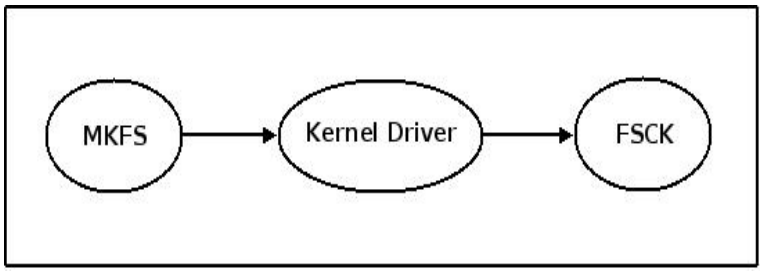

Figure 3.4: *Filesystem development phases*

Since we have used the ext2 filesystem to implement our feature, the three parts of it which we have concentrated on in this work are *mke2fs*, *ext2 driver*, *e2fsck*. Each one of these parts is essential to the others, that is, each one of them complements the others. These implementations will be discussed shortly.

3.3.1 Mke2fs Implementation

Mke2fs is the crucial tool which the operating system uses interactively with the user to decide the filesystem layout on any sort of block devices. During the course of its operation, many features can be enabled which have an impact on the performance of the filesystem

The design proposed for the *chunks* feature is implemented in *mke2fs* and enabled during the creation time through a specific incompatible list of features of the ext2 filesystem. During creation time the user can instruct the mke2fs to enable the *chunks* feature using the following syntax:

```
$ mke2fs -O chunks /dev/hda1
```

From the above syntax the *-O* switch is used to specify the features which are requested to be included in the filesystem. Our *chunks* feature is then paired with the filesystem set of incompatible features which includes the EXT2_FEATURE_INCOMPAT_CNODES feature that matches ours. Once enabled, the feature is used in the kernel-wide environment as a key to instruct the kernel to use the functionality that comes with the *chunks* feature.

3.3.2 Kernel Driver Implemenation

The kernel driver implements the means through which the virtual filesystem (vfs) deals with the filesystem, that is, the vfs functions are mapped to the functions of the filesystem driver. The chunks feature provides further functionalities to the filesystem, which enables it to make use of the existing layout of the disk.

This feature is mainly implemented by the *cnode*-allocation algorithm. The cnode-allocation algorithm strives to implement a new allocation strategy inspired by chunkfs continuation inode implementation. This algorithm uses the basic search strategy of the ext2 filesystem.

When a regular file expands due to a write or truncate operation, the kernel attempts to allocate extra data blocks for the file in question. In order to accomplish this task, the kernel invokes the *ext2_get_block* function. This function handles all the required operations which pertain to data blocks allocation. The most important search and allocation operation, which is invoked by *ext2_get_block*, is the *ext2_alloc_block* function. This function attempts to allocate a new data block next to the last data block allocated to the file.

If the preallocation of blocks is supported by the file system, the filesystem uses the data blocks which were preallocated in the previous call to *ext2_alloc_block* function. If no preallocated data blocks are available, the function attempts to allocate a new block in the same blockgroup of the expanded file (which include the file's inode).

If the previous allocation operation fails, the last resort is to call *ext2_new_block* function to search for a free block in the vicinity of the filesystem to allocate a new data block in another blockgroup. When a blockgroup which has both free inodes and data blocks is encountered, and if the kernel supports the cnodes feature, that is EXT2_FEATURE_INCOMPAT_CNODES is enabled (this feature is enabled in the *mke2fs* time), the kernel invokes *ext2_new_cnode*, otherwise the kernel performs the default block allocation mechanism.

The *ext2_new_cnode* function takes two arguments: the *parent* inode, that is the file's

inode for which we want to allocate a new data block, and the *mode* of that inode (i.e regular file, directory). If the *mode* field of the parent inode is neither a regular file nor a directory (i.e. pipe), the function aborts, failing to allocate a new inode. On the other hand, if this is not the case, *ext2_new_cnode* allocates a new inode in the blockgroup which has both free inodes and data blocks. We are emphasizing the case of allocating a *cnode* in a blockgroup which has both free inodes and data blocks because the newly allocated *cnode* will require free data blocks and these data blocks will be counted toward the free blocks of the *parent* inode.

The new allocated inode shall be called a *continuation inode* or *cnode*. This is exemplified by the implementation of the *ext2_new_cnode* algorithm. To illustrate this point, the new allocated inode will be considered a child to the parent inode; as a result, the child inode is linked to the *parent* inode data structure resulting, in a continuation to the file's inode expansion, hence the name "cnode." The number of allocated cnodes can be adjusted as required; however, the number of allocated cnodes should conform to an important consideration, that is, the disk fragmentation reduction. The data structure which is used to attach *cnodes* to the *parent* inode is a dynamic link list, where the parent represent the "root" and the *cnodes* represent its direct children.

According to the previous procedure, the kernel may also attempt to allocate a *cnode* to the last allocated *cnode*. In this case, instead, *ext2_new_cnode* checks if the *parent* (the *cnode* in this case) has the EXT2_CNODE_FL enabled (i.e. is a *cnode*). In this case, instead of allocating a new *cnode* for the current cnode, the algorithm backtracks to the *parent* inode of the current *cnode*, and attempts to allocate a new *cnode* and link it to the parent inode if the permissible number of cnodes allowed is not met yet.

The above implementation exemplifies the concept of "chunks." As noted previously, the only case when a file has to allocate a new block in another blockgroup is when it expands due to a write or a truncate operation. In such case we considered the blockgroup as a *logical chunk*, since the linkage (logical link) among the blockgroups is done through the

cnodes list and as illustrated in figures 3.1 and 3.2. This implementation extends the *inode* data structure of the ext2 filesystem by having three new fields:

1. *i_parent*: the parent inode of this inode. Usually it is the inode of the file which needs to expand due to a write or truncate operation.

2. *i_cnodes_list*: head of the list which contains the cnodes list structure. Basically the head of this list is the *i_parent*.

3. *i_cnode_count*: this field is used by the parent inode and its children *cnodes* to determine if they have exceeded the allowed number of *cnodes*. For the purpose of testing, this number is set to EXT2_NUM_CNODES (4).

The cnodes-allocation algorithm is illustrated below:

3.3.3 E2fsck Impelmentation

Fsck is the filesystem checker which is used in almost all of the Unixes for the purpose of checking the consistency of the filesystem and the coherence of its structure. The checker's function is to repair and bring back the filesystem components into a consistent state after an unclean mount, power and hardware failures, or a filesystem crash.

The historical implementation of fsck [15] discovers the inconsistencies in the superblock structure, specifically the file-system size, number of inodes, free-block count, and the free-inode count. It also checks the inode state, specifically the inode format and type, link count, duplicate blocks, bad blocks, and the size of the inode. The filesystem checking goes through five default phases or passes which are performed sequentially and an additional phase which is introduced for the purpose of our implementation. The phases of interest regarding the implementation of the *condes* feature on the *e2fsck* are explained here, and other phases are mentioned in brief.

- **Pass 1 – Blocks and sizes checking**

Algorithm 1 ext2_new_cnode: cnodes allocation algorithm

```
EXT2_ALLOC_BLOCK(parent, ..)
  .
  .
  .
  for each (Block group in the ext2 partition)
    Find free inodes
    Find free data blocks
    if  (free inodes and free blocks)
      EXT2_NEW_CNODE(parent, parent → mode)


EXT2_NEW_CNODE(parent, mode)
  if ( (mode = REG_FILE) or (mode = DIR) )
   retry:
    if ( parent → flags = EXT2_CND_FL )
      parent ⇐ parent → parent
      goto retry
      if ( parent → count < EXT2_NUM_CNODES )
        inode ⇐new(inode)
        initialize(inode)
        inode → flags ⇐EXT2_CND_FL
        parent → cnodes ⇐ add_entry(inode)
        parent → count ⇐ count = count + 1
      else
        return nospace
```

During this phase the inodes of the filesystem are checked for consistency, that is, checking the type and state (inode mode), link count, duplicate blocks, bad blocks, and inode size for each inode. For the inode mode which specifies the inode type and state, there are six types of inodes [15] : *regular inode*, *directory inode*, *symbolic link inode*, *special block inode*, *special character inode* and *socket inode*. In addition to these, a new inode type, *cnodes*, is introduced. Thus in the first phase of checking, all these inodes which are checked and appear to be of type *cnode*, i.e have an EXT2_CNODE_FL flag enabled, are stored in a special list *cnodes_list* and their inode numbers are marked as used in a newly introduced bitmap (*inode_cnode_map*) in order to be checked in a later phase (pass 6). That also includes the checking of the inodes which has *cnodes*, i.e., has an EXT2_HAS_CNODE_FL flag enabled. These are stored in a special new bitmap (*inode_has_cnode_map*) in order to be checked in phase (pass 6). These bitmaps differ from the default bitmaps which are created in the case of ordinary non-cnodes filesystems.

- **Pass 2 – Check path names**

  This phase checks the consistency of all the directories in the filesystem and their descended entries. It should be clear here that since the *cnode* is not a *directory* inode, then only the default checking takes place.

- **Pass 3 – Check connectivity**

  This phase checks the directory connectivity to the "\" directory. Since the *cnodes* are basically attached to inodes in the same chunk (blockgroup), it should be obvious that it tracesback to the "\" directory.

- **Pass 4 – Check Reference counts**

  This pass concerns itself with link count of the inodes seen in pass 2 and 3. Again since the *cnodes* are not directory inodes, which implies that they do no have directory entries, then its link count is 1. Moreover, if the inode read appears to be disconnected,

we first check whether it is a *cnode*. If this is the case, we skip pass 4, as we will deal with this case in pass 6.

- **Pass 5 – Check Bitmaps**

  This pass is concerned with checking the block and inodes bitmaps for consistency with those on disk.

- **Pass 6 – Cross-chunk check** This pass is an extra proposed pass which follows the previous ones. This pass is incorporated to check the spanning of *cnodes* over the blockgroups by checking their connectivity. The implementation incorporated here is an optimized version of the implementation proposed in *Chunkfs* [3] .

  This pass depends on the information collected in previous passes, namely *cnodes_list*, which contains a list of all the *cnodes* in the filesystem which are encountered in pass1. Moreover, this pass also depends on the information collected in both the *inode_cnode_bitmap* and *inode_has_cnode* bitmaps.

  This pass also verifies all these inodes in the partion which are *parent* inodes by verifying them against the *inode_has_cnode_map* bitmap. Once figured, all the *cnode*s which pertain to this parent inode are checked through verifying the parent's cnodes list by comparing them to the list of collected *cnodes* and the *inode_cnode_map* bitmap.

  The entries stored in the *cnode_list* are of type *cnode_info* structure. This structure is introduced to store the *cnode* itself along with other accounting information in order to facilitate the recognition of the cnodes in this pass. The structure includes information such as the cnode itself, inode number and a link to the other *cnode_info*

  One of the main considerations in this pass is how the cnodes checking is handled. The main assumptions when checking the parent's list is that all the entries of the cnodes list collected in Pass1 represent all the cnodes in the filesystem. Thus, the cnodes attached to a particular parent should exist in that list. If for any reason the cnodes do not exist in that list, then the pass abort with an error message.

Chapter 4

Chunks Feature Performance

In order to point out the virtues of the chunks feature, two main a priori considerations were taken into account:

- According to [22] , most of the files on productions machines tend to be small in size. Thus the likelihood of having a large number of continuation inodes is minimal.
- The checking time for the filesystem metadata integrity depends on the size of the files contained in the filesystem.

Since the work follows *Chunkfs* goals in achieving a minimum checking time, these two main points were considered when the chunks feature was evaluated, that is, the minimization of filesystem checking time depends on the existence of cross-blockgroups (chunks) links which are determined by the files sizes. These links should reduce the amount of consumed time taken to check the integrity of filesystem metadata.

However, the occurrence of the above two points is mutually exclusive, that is, whenever the filesystem contains files which have on average small sizes, the number of cross-blockgroups links is small. Whereas, whenever the filesystem contains files which have on average a large size, the number of cross-blocksgroups links is high.

Based on the above hypothesis and estimations and from our initial expermentation, we were able to prove that the files with smaller sizes tend to have the minimum number of cross-blockgroup links (cnodes). According to our cnode-allocation algorithm, only 0.04%

of the filesystem files had cross-blockgroup links. The main reason for this is the tendency of the small files to favor the usage of the first 12 data blocks and the remaining three blocks for simple indirection whenever there are enough data blocks in the file's local blockgroup.

Moreover, as the main goal for this study was to prove that the filesystem checking time was reduced, the experimental findings, which were based on tangible evidence, proved that point.

The approach of finding how many blockgroups were verified during the fsck checking was followed. According to that, the experiments showed that for our dataset which occupies a 15GB filesystem partion, an estimate of approximately 96.2% of the filesystem needed to be checked. Despite the obtained statistics, these results depend on the size of the dataset used, which is disscussed in the next section.

4.1 The Data Set

The chunks feature impelmenation is based on the ext2 filesystem implemenation. For that matter, the filesystem was chosen with the default block size (1024 bytes) to be installed on the 15GB partition. The default inode size of (128 bytes) was chosen, thus the sizes of the inode tables and the data blocks for each blockgroup are based on these default sizes and hence are counted towards the eventual size of the blockgroup.

The dataset used is files from a server machine with mostly ASCII files. Some other file types, such as image files and movie files, were used to reflect a diverse collection which can be found on any type of machine and personal computers. The dataset was duplicated properly to fill the filesystem, with some free space left. Thus we have formed a fair distrubtion of files with different sizes and types.

In Figure 4.1, the distribution of files is shown. This distribution of files was chosen such that the filesystem benchmarking is used with a realistic dataset, such as the one used in [22] with file size of about 98% for files with size mostly smaller than 512 KB and 2% for files with a size of more than 1MB.

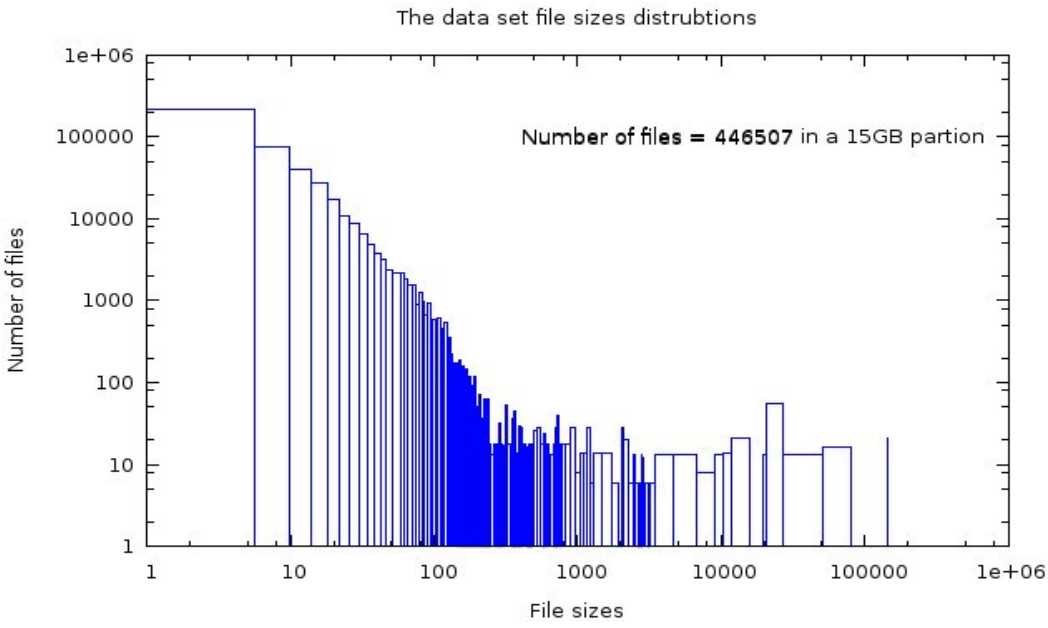

Figure 4.1: *Distrubtion of files in the dataset*

4.2 Filesystem Development Toolkit

The modified e2fsprogs which provides both *mke2fs* and *e2fsck* was used to setup and check the filesystem. For the experimental kernel, User Mode Linux (UML) [23] was used. UML provides a virtual machine which simplifies the development, the testing of new features, and the debugging of the kernel code and distributions. UML enables the user to run an entire distribution tree and linux image as a normal process under the current kernel and mount it as required. The distribution tree image can be mounted using a loop back device.

With GNU's GDB [24] , the usage of UML proved to be quite convenient as it was easier to re-run the experimental kernel multiple times without affecting the user machine, and to test it, and debug it by interrupting kernel functions calls just like any other normal program. Also the *dd* [25] command was used to create sample non-block device partitions for the sake of testing by creating custom-sized data files with null data.

Files Size(KB)	*Number of Files*	*Percentage*
$0 - 512$	445659	98%
$512 - 1$ (MB)	376	0.08%
$1 - 10$	316	0.07%
$10 - 100$	134	0.03%
$100 - 1000$	22	0.004%

Table 4.1: *Dataset Files Sizes and Percentages*

4.3 Estimated Effort Involved

As mentioned in section 3.3, modifications were made to three parts of the filesystem. Most of the research time was spent on reading the kernel source code and research journals which pertain to filesystems. The time spent on reading the kernel source code was essential to determine where the changes should be made.

The estimated number of lines of code is 4KLOC. This includes the lines of code in the filesystem creation and checking tools (mkfs and fsck) and the lines of code from the kernel itself. In contrast to the previous implementation [3] , which reimplemented and heavily changed the structure of the used filesystem, the current implementation used the underlying functionality (e.g. kernel link-list implementation) provided by the kernel without the need to reimplement it.

4.4 Experimentation

The experimentation approached followed for testing the chunks feature is close to the approach used for *Chunkfs*, with a number of variations. These variations are governed by the alternative design of the chunking feature and the continuation inodes that differ essentially from that in the predecessor implementation.

4.4.1 Continuation Inodes Number

As stated in the beginning of this chapter, we considered an important trade-off between the number of continuation inodes and the file sizes. We used a dataset in which there is 98% of files had sizes less than 512KB and only 2% of files had more than that. This is illustrated in table 4.1 and Figure 4.1

An important point should be considered here before moving forward with the results. The number of free inodes in the filesystem blockgroups (chunks) was decreased more than the norm due to the increase in the percentage of inodes allocated by the cnode allocation algorithm from the other blockgroups, where the file's inode is not located. Although this might be considered a design drawback, we hoped that the rapid increase in disk size [1] would alleviate this fact, such that the increased disk sizes can motivate the increase in the default block size [1].

Per our initial expectations regarding the block and inode sizes and according to the number of free data blocks, we have found that files with small sizes tend be the files which are less likely to span across multiple blockgroups (chunks) thus minimizing the number of required continuation inodes. From 446507 files in the partition, only 188 files have continuation inodes and the remaining do not. Thus, the obtained percentage was only 0.04% of the number of total files with continuation inodes; and the remaining 99.96% have normal files as shown in Figure 4.2.

Despite the very small percentage of files with continuation inodes, we opted for a better comparison method. This method implies the exclusion of the large percentage of files which do not have continuation inodes as these form the major percentage of files in the partition, and the inclusion for comparison of those files which have both continuation inodes and approximate sizes, i.e., the 0.04% of files with sizes greater than 1MB. The figure 4.3 depicts this parentage of files which are both greater than 1MB in size and have continuation

[1] The default 1024-byte size block can store up to 8 inodes of the default size (128-byte), whereas the 4096-byte block size can store up to 32 inodes

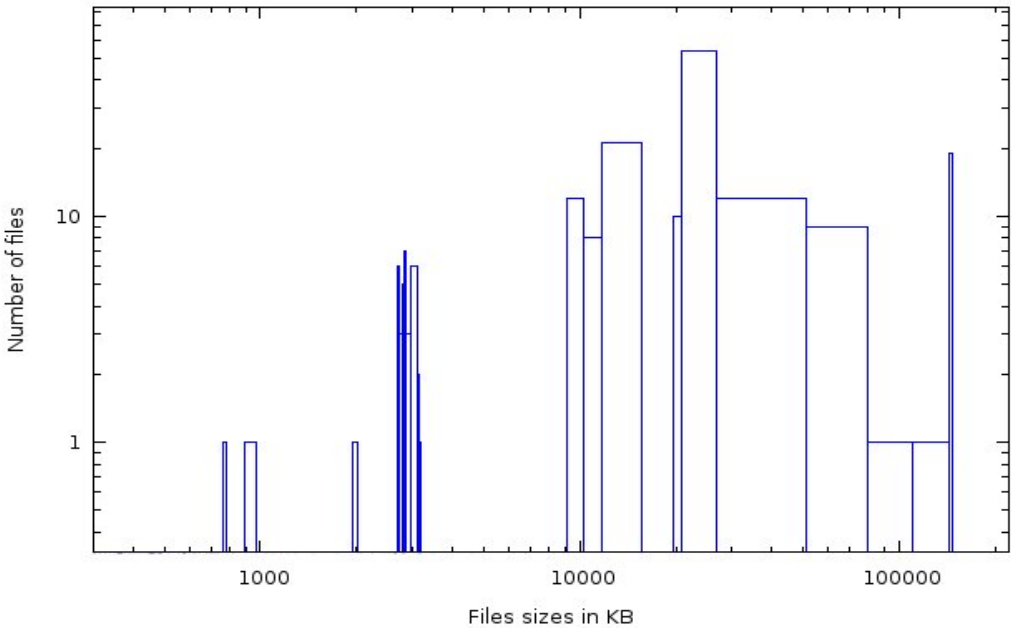

Figure 4.2: *Distribution of files with cnodes*

inodes.

The figure 4.3 improves our view of the correct percentage distribution of the files with continuation inodes. It shows that of the percentage of 0.04% of files with sizes greater than 1MB, 39% of the files have continuation inodes, whereas the remaining 61% of the files do not.

Based on the 39% of files with continuation inodes which was obtained previously and as shown from the figure 4.3 which is based on that percentage, it is clear that the files which are greater in size used more continuation inodes than those which are smaller in size. The number of continuation inodes (recall that it was a maximum of 4) used increased as the file size increased. The files with sizes of more than 512KB and less than the 10MB used between 1 and 2 and rarely 3 continuation inodes, whereas files which are greater than 10MB in size usually used all their share of continuation inodes. Surely, since the snapshot of files used in the experimentation was a snapshot of files in a server machine (which are mostly ASCII files), we hoped that the size of files is not large, so as to allow the increase

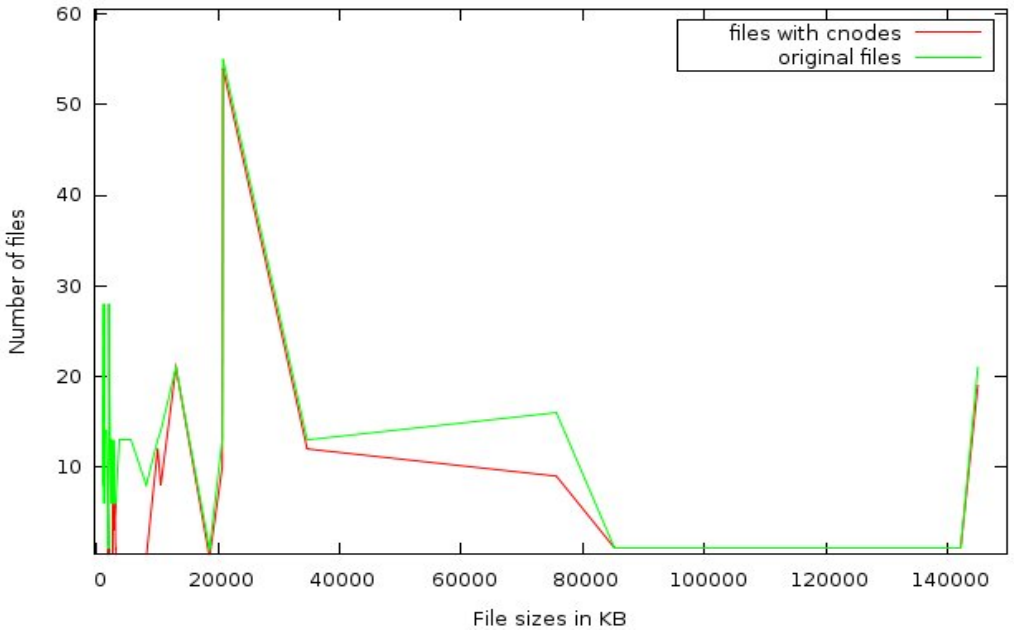

Figure 4.3: *Improved distribution of files with cnodes*

in the number of continuation inodes[2].

It should be taken into consideration that these chart graphs and the percentages obtained are subject to the main metrics of block size, inode size, the eventual experimental partion size, and the file sizes contained in each partition. However, we can almost conclude that as file sizes increase, the number of inodes utilized as continuation inodes is also increases. The number of continuation inodes assigned to each file's inode can also be relaxed to use more than 4 inodes as continuation inodes; this, however, will affect the usage of the number of free inodes in the filesystem, unless the block size is increased accordingly.

4.4.2 Fsck-time Reduction

As mentioned in the beginning of the chapter, for the dataset which occupies most of the 15GB partition, we followed the approach of checking how many blockgroups were dirty after the crash. This includes checking the linkages of all the continuation inodes.

[2] For the sake of experimentation we overlooked the fact that most of the server machines use huge size databases which can host large amount of data; however, most of these databases have their own methods of dealing with filesystem to best utilize their data storage.

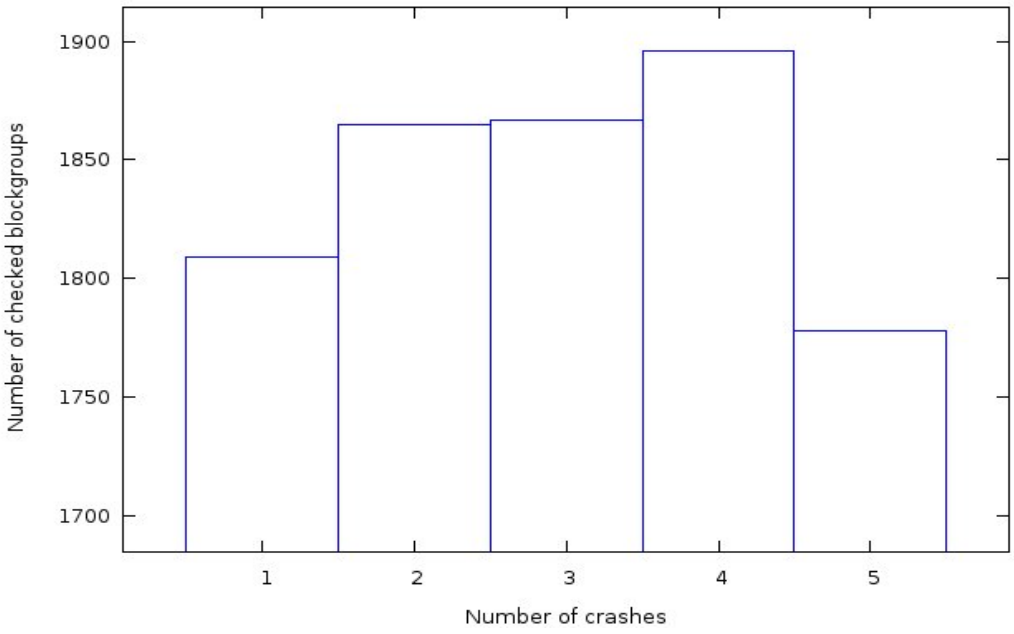

Figure 4.4: *Number of blockgroups checked per crash*

We intentionally mounted the filesystem and halted it instantly without unmounting so we could obtain a crashed filesystem with an inconsistent state. For the five times we did that, we checked the filesystem with the modified fsck tool which we have developed.

As shown from Figure 4.4, for the total number (1920) of blockgroups checked during the five crashes, about 96% of the blockgroups which were simultaneously dirty during these crashes were checked, leaving only 4% as clean. Thus the filesystem checking time was reduced by approximately 4.3% of the normal time required to check the filesystem.

Another experimentation was achieved in terms of *fsck time crunch* [3] . The experimentation considered the comparison between the time *fsck* had taken in the case of the included modifications and the time it had taken in the case of its normal operation. Figure 4.5 depicts five checking operations which were performed after five unclean halts (crashes) of the filesystem. During these checking operations, we observed a varying percentages of dirty blockgroups (chunks). For all of the previous operations, on average, we found that approximately 95% of the checking time was consumed in performing the entire experiment, leaving us with approximately only 5% reduction in the checking time.

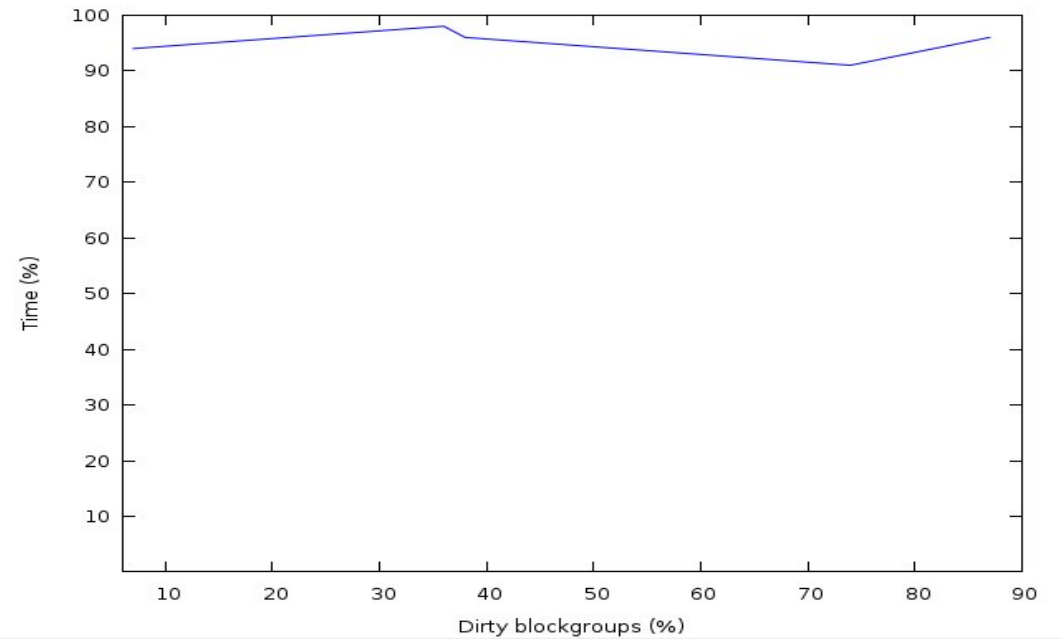

Figure 4.5: *The measured checking time*

During the different experimentations, we noticed that the obtained percentages were not quite significant. However, this may argue for better experimentations. For example, the filesystem partition size and the dataset size could be increased to obtain different results. Also different inode and block sizes can be used during the experimentations. More optimized techniques for locating and checking the dirty blockgroups could also be used for this work.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The recovery-driven design is an open problem for continued research and experimentation to improve filesystem performance in order to cope with the rapid constant increase in disk size, as we pointed out in previous chapters. Despite the many attempts to avoid the time-consuming procedure of checking filesystem integrity, in certain situations it becomes inevitable to consider such checking. Thus, filesystem designs should be probed further to provide better utilization for disk space usage; yet a smarter design can prove its usefulness when it comes to filesystem inconsistency check. This design should cope with the challenges of the next decade.

From the experimental evaluations done with Chunks feature, we found there could be room for improvement when it comes to file system checking time. This improvement is seen by considering blockgroups as fault-isolated components with controlled connectivity to other blockgroups. Thus this has an effect on improving the entire filesystem durability, resistance to corruption, improving disk usuage despite its excess use of disk inodes, and most important, minimizing the cost of checking time.

5.2 Future Work

Our long-term goal is to mature the feature implementation enough in order to be eligible for inclusion in the experimental design of the emerging EXT4 filesystem. However, more improvement into the continuation inodes allocation algorithm can be taken into consideration [26] . Other attempts include implementing the chunks feature directly into the VFS layer of the operating system.

Bibliography

[1] M. H. Kryder, "Data-storage technologies for advanced computing," vol. 257, pp. 116–125 (Intl. ed. 73–81) (Intl. ed. 72–??), Oct. 1987.

[2] S. Best, D. Gordon, and I. Haddad, "Kernel korner: Ibm's journaled filesystem," *Linux J.*, vol. 2003, no. 105, p. 9, 2003.

[3] V. Henson, A. van de Ven, A. Gud, and Z. Brown, "Chunkfs: using divide-and-conquer to improve file system reliability and repair," in *HOTDEP'06: Proceedings of the 2nd conference on Hot Topics in System Dependability*, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2006.

[4] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: a solution to the metadata update problem in file systems," *ACM Trans. Comput. Syst.*, vol. 18, no. 2, pp. 127–153, 2000.

[5] M. K. McKusick, "Running "fsck" in the background," in *BSDC'02: Proceedings of the BSD Conference 2002 on BSD Conference*, (Berkeley, CA, USA), pp. 7–7, USENIX Association, 2002.

[6] D. P. Bovet and M. Cassetti, *Understanding the Linux Kernel*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2000.

[7] S. D. Pate, *UNIX Filesystems: Evolution, Design, and Implementation*. Wiley, 2003.

[8] C. Mason, "Journaling with reisersfs," *Linux J.*, p. 3.

[9] H. T. Reiser, "ReiserFS." http://www.namesys.com.

[10] H. Reiser, "Kernel korner: trees in the reiser4 filesystem, part i," *Linux J.*, vol. 2002, no. 104, p. 8, 2002.

[11] T. T. Card, Remy and S. Tweedie, *Design and implementation of the second extended filesystem*. 1994.

[12] T. Y. Ts'o and S. Tweedie, "Planned extensions to the linux ext2/ext3 filesystem," in *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 235–243, USENIX Association, 2002.

[13] T. Ts'o, "E2fsprogs." http://e2fsprogs.sourceforge.net/.

[14] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix (revised july 27, 1983)," tech. rep., Berkeley, CA, USA, 1983.

[15] T. J. Kowalski, "Fsck—the unix file system check program," pp. 581–592, 1990.

[16] L. W. McVoy and S. R. Kleiman, "Extent-like performance from a UNIX file system," in *USENIX Winter*, pp. 33–44, 1991.

[17] M. K. McKusick, "Running "fsck" in the background," in *BSDCon* (S. J. Leffler, ed.), pp. 55–64, USENIX, 2002.

[18] M. Wu and W. Zwaenepoel, "eNVy: A non-volatile, main memory storage system," in *ASPLOS*, pp. 86–97, 1994.

[19] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham, "The episode file system," in *Proceedings of the USENIX Winter 1992 Technical Conference*, (San Fransisco, CA, USA), pp. 43–60, 1992.

[20] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 1–15, ACM, 1991.

[21] T. T. Val Henson, Zach Brown and A. van de Ven., "Reducing fsck time for ext2 file systems," in *The Linux Symposium*, 2006.

[22] A. S. Tanenbaum, J. N. Herder, and H. Bos, "File size distribution on unix systems: then and now," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 100–104, 2006.

[23] J. Dike, *User Mode Linux*. Bruce Perens Open Source series, 2006.

[24] "gdb debugger." http://sourceware.org/gdb/.

[25] "Take command: What is dd?," *Linux J.*, p. 11.

[26] K. A. Smith and M. Seltzer, "A comparison of ffs disk allocation policies," in *ATEC '96: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 1996.