UNIVERSITY of CALIFORNIA
SANTA CRUZ

CLEANING THE CLEANER FOR THE LINUX LOG-STRUCTURED FILE SYSTEM

A dissertation submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

David A. Gatwood

June 2001

The dissertation of David A. Gatwood is approved:

Professor Tara Madhyastha, Chair

Professor Glen Langdon

---

undefined

Vice Provost and Dean of Graduate Studies

Copyright © 2001

David A. Gatwood david@gatwood.net

All Rights Reserved.

This material may be freely distributed for non-commercial purposes subject to the terms and conditions set forth in the Open Publication License v1.0 (http://opencontent.org/). Commercial distribution in whole or in part is prohibited without the prior written permission of the author. Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Contents

List of Tables **iv**

List of Figures **v**

1 Introduction **1**

2 Related Work **5**

3 Implementation **7**
3.1 General Issues 7
3.2 Special Workarounds 10

4 Results **12**
4.1 Andrew Benchmark 13
4.2 Kernel Benchmark 15
4.3 PostMark Benchmark 16
4.4 Analysis 17

5 Known Problems **19**

6 Conclusions **22**

7 Acknowledgements **24**

Bibliography **27**

List of Tables

4.1 Andrew Benchmark. 14
4.2 Kernel Benchmark. 15
4.3 PostMark Benchmark. 17

List of Figures

1.1 A mostly-dead log entry. 3

3.1 User-kernel Separation. 9

7.1 Mr. Penguin in the wild. 24

Chapter 1

Introduction

Time is an illusion; lunchtime, doubly so.
– *Ford Prefect*

A log-structured file system is a file system which, unlike traditional file systems, writes data in a "log" by appending to the end of that log. This is conceptually similar to logging bank transaction data on magnetic tape. When one tape gets full, you continue appending by adding another tape. Obviously this makes no sense for a disk, since the cost of adding disks every few days is orders of magnitude more expensive than adding tape. For this reason, the log structure is approximated by dividing the log into "segments" of fixed length, which can then be freed if they no longer contain any relevant data. This file system design is presented in more detail in the seminal LFS paper [13].

In the last decade, much attention has been given to the design and implementation of log-structured file systems. The use of a log structure offers numerous

advantages over a traditional file system, including elimination of small write penalties in RAID systems [10], diminished file system check delay in HA systems, and more recently, better wear-leveling in flash file systems [8].

Most of the effort in this area, however, has been focused on implementing an LFS for systems based upon BSD UNIX. The reasons for this are threefold. Source code for BSD is freely available, making implementation of file systems much easier than it might be in a more proprietary form. The license used by BSD allows additions to remain proprietary, meaning that spin-off companies can be created to sell technology based upon BSD research. Finally, BSD has traditionally been one of the more popular variants of UNIX. BSD derivatives include SunOS, BSDI, and Darwin, to name a few.

However, the license used by BSD, while making it a particularly popular choice for research, also prevents distributing these file systems as part of Linux. Linux is an operating system whose installed base is believed to be among the largest of any UNIX variant [14]. It is thus considered very desirable to have a log-structured file system for Linux.

In 1998, Christian Czezatke created a log-structured file system for Linux, known originally as dtfs [4] and later as LinLogFS. This file system broke new ground for Linux file systems; however, one crucial piece was missing: a cleaner. In log-structured parlance, a cleaner is a program that takes log segments that contain blocks from deleted files or old copies of existing file data and combines the remaining portions into a new, "clean" log segment. 1

Without a cleaner, a log-structured file system is of limited utility. When the

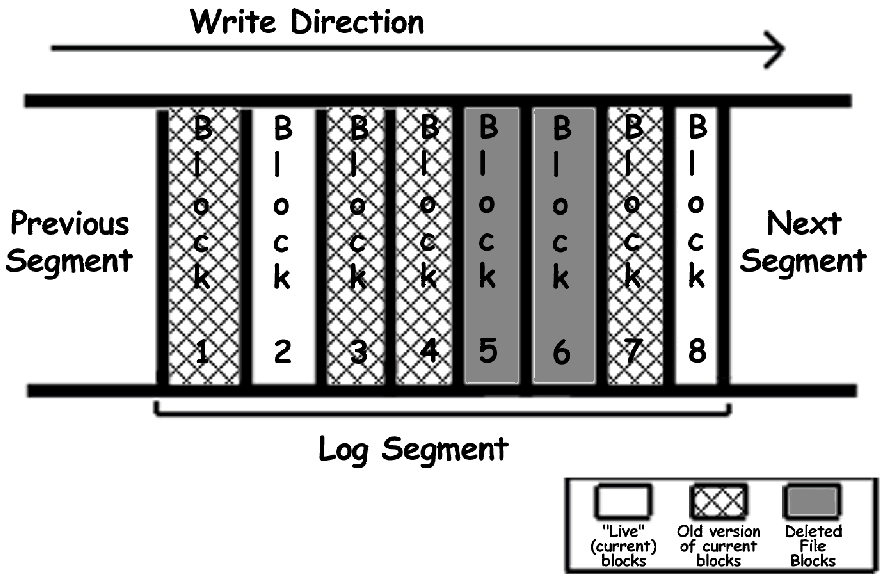

Figure 1.1: A mostly-dead log entry.

log size exceeds the total size of the partition or disk, no more data can be appended to the log, even if no active data remains in the file system. Furthermore, performance of an LFS without a cleaner has little meaning, because the cleaner introduces a performance penalty.

The goal of the LinLog Cleaner Project (LLCP) is to create a working cleaner. The cleaner is now functional and is sufficiently stable to allow performance testing under moderate loads. Some additional work is necessary before this implementation will be suitable for long-term use in a production system.

In Section 2, we discuss previous previous LFS research and give a potted history of LinLogFS. In Section 3, we explain the implementation of the cleaner and

some of the issues involved in its completion. In Section 4, we evaluate LinLogFS using various file system performance benchmarks. In Section 5, we describe remaining problems that we have yet to resolve. Finally, in Section 6, we discuss what we have accomplished and explain how to obtain source code.

Chapter 2

Related Work

> For every complex problem, there is a solution that is simple, neat, and wrong.
> – *H. L. Mencken*

Log-structured file systems have been available for many operating systems, such as NetBSD and other 4.4BSD derivatives, for many years. These are derivatives of BSD-LFS [12], written by Margo Seltzer for 4.4BSD. This research was influential in the design of the Linux Log Structured File System, LinLogFS, upon which our work was based.

Several researchers have designed cleaners for various log-structured file systems. Trevor Blackwell [2] wrote a paper on a heuristic-based methodology for segment cleaning, designed for scalability. However, LinLogFS in its original state lacked sufficient kernel support for the cleaner to make this a practical choice.

We instead adopt a more modest approach, building on the work of a team of programmers from the University of Wisconsin, known as MrClean [11]. The MrClean

project gave us working code for the user-space portion of the cleaner (apart from some changes to make the code more environment-neutral). This allowed us to focus on the more difficult matter of kernel support for the cleaner.

Fortunately, the MrClean project also provided us with some (largely non-functional, but conceptually sound) kernel code. The majority of our effort was directed towards making that kernel code work correctly, and to that end, this effort was successful.

Chapter 3

Implementation

> The generation of random numbers is too important to be left to chance.
> – *from UNIX fortune*

The cleaner is implemented with one part sodium hydroxide to three parts water, with possibly a touch of a glycol compound added for good measure. Oh, wait. You mean the file system cleaner....

3.1 General Issues

The cleaner can be divided into two parts: the kernel portion and the user-space portion. The user-space portion of the cleaner does a relatively small part of the work involved in cleaning a file system. Specifically, it asks the kernel portion for information about each segment (usage flags for the constituent blocks), and then decides whether it has enough wasted blocks to warrant the effort of creating a new log segment for its data, or whether it should simply leave it alone.

If the cleaner decides to "refile" the blocks in that segment, it adds that segment to a list, and later makes a kernel call to refile each block. Finally, after exhausting the list of blocks to be refiled, it syncs the file system, and quits.

The kernel portion of the cleaner is somewhat more complex, because it modifies a live file system without using the standard file system interfaces, in effect changing it right under the VFS layer's nose, as shown by Figure 3.1. The dashed line through the VFS layer represents an ioctl() on the .ifile file which the VFS layer passes directly to the underpinnings of LinLogFS.

Fortunately, the original cleaner authors implemented some locking to prevent a refile from being concurrently committed with a file system write. This locking, coupled with inode versioning in the unified buffer cache, guarantees a consistent view of the file system to readers.

The kernel support for the cleaner consists of a function to refile an arbitrary log block and various subfunctions that it uses. The basic process begins by checking the log block to determine which physical blocks it contains. For each physical block, the refile code then determines which inode is associated with that physical block, and reads the inode. If the block is not one of the first few blocks in the file then the refile code will read indirect blocks as needed.

Next, the refile code compares the log block number to be refiled to the corresponding log block number in the in-core copy of the inode or indirect block from which it is referenced. If the log block numbers match, then this is the latest version of the block, and it must be refiled. If they do not match, then this data block is stale,

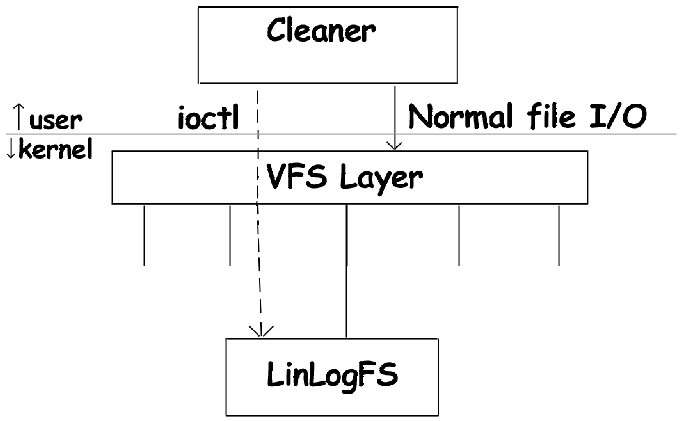

Figure 3.1: User-kernel Separation.

and can be discarded. Next, the code begins marking blocks dirty.

Although it is clear that the data block must be marked dirty, the buffer pool in Linux is designed in such a way that when changes to a file system get synced to disk, the in-core copy of any written block is freed. If an indirect block has a memory reference to that block, then that indirect block must also be considered dirty, because that memory reference will now be a dangling pointer.

This applies not only to the indirect block nearest the data block, but also to any indirect block that references that indirect block, where applicable, and to the inode itself. This is an unfortunate side-effect of the design of the Linux buffer cache,

at least in 2.2.x kernels.

Finally, after marking the various parts of the in-core copy of the inode, indirect blocks, and data blocks dirty, the file system changes are flushed to disk, and the syscall returns, allowing the user-space portion of the cleaner to clean the next block, and so on.

3.2 Special Workarounds

We encountered several design issues that impacted the stability of the cleaner. A LinLogFS file system contains several "special" files, located at the root of the file system. These files are called .atime, .ifile, and .iusage. These files do not actually exist anywhere on disk. They provide direct access to data structures internal to the file system itself, for the purposes of enabling cleaning.

These special files can be read using the usual methods, but they cannot be written in the same way because they do not actually have inodes. An iget returns a fake inode structure, but an iput is silently ignored. For this reason, files with "fake inodes" cannot be cleaned in the normal way.

Ideally the blocks in these files should be cleaned, but this would require a substantial increase in code volume with the potential for many new bugs. In the interest of making the file system work without destroying data, we deemed such expansion excessive, so we elected to make any attempt to clean blocks in these files return an error value. The net result of this change is that blocks in the special files

may end up unnecessarily taking up entire segments. However, the affected files are sufficiently dynamic (and sufficiently small) that leaving them alone is unlikely to waste very many log segments. This issue is described in more detail in Section 5.

Chapter 4

Results

To you two, therefore, will I unfold the truth as it shall be.
– *Homer.*

To evaluate our changes to LinLogFS, we performed several performance tests comparing LinLogFS to ext2 for three benchmarks. We conducted these experiments on an 800 MHz Pentium III machine with an ATA/33-compliant hard drive, and 256 megabytes of RAM. We used 2 gigabyte partitions for both file systems. We used a single-user workstation to minimize the chance of other tasks impacting performance. We reinitialized these partitions with a clean file system before running each set of benchmarks to prevent any test from skewing the results of any other test.

We chose the ext2 [3] file system for comparison because it is the standard file system for Linux workstations. Ext2 is derived from the ext file system, and both are conceptually similar to ffs [9] and other traditional UNIX file systems. Also, LinLogFS is implemented as a log-structured wrapper around ext2, making this a good measure

of the impact of log structuring on an existing file system.

4.1 Andrew Benchmark

The first test suite we used was the Andrew Benchmark [6]. Carnegie Mellon University used this series of file system benchmarks to test the performance of the Andrew File System [5]. The Andrew Benchmark shows that ext2 is more than twice as fast as LinLogFS. These results can be found in Table 4.1.

While performing this test, we discovered a bug that causes file system corruption when file writes and block refiling (cleaner activity) are allowed to happen concurrently. As a workaround, we added wrappers around the file write and block refile functions to prevent concurrency. We present both the results with and without the added locking. In Table 4.1, linlog-nl represents the version without locking and linlog-nlnc represents that version without concurrent use of the cleaner. The remaining benchmarks were performed only with locking turned on for stability reasons.

The Andrew benchmark has five phases. Phase I involves creating twenty directories. We expect this phase to complete in a very short time on most modern file systems. Phase II copies about 200 megabytes of data into the file system, spread across 70 files. This tests file system throughput, and is expected to take much longer than phase I. Phase III does a recursive "ls -l", effectively performing a stat on every file and directory created in the previous two phases. We expect this phase to be relatively fast on most modern file systems, comparable to phase I. Phase IV recursively greps

description	Phase I Phase IV	Phase II Phase V	Phase III Total
ext2	0.0	0.0	0.2
	0.2	1.8	2.2
linlog	0.4	1.4	0.2
	0.4	2.2	5.0
linlog-nl	0.4	1.4	0.6
	0.2	2.0	4.6
linlog-nlnc	0.4	1.4	0.2
	0.4	2.2	4.6

Table 4.1: Andrew Benchmark.
(times are in seconds)

for a simple expression in the files created in phase II. This is fairly read-intensive, and we expect it to take longer than any of the previous phases. Phase V compiles a small X11 windowing manager. This phase is also read-intensive, but the reads are interspersed with periodic writes as object files are generated. We expect this to be the longest part of the test because it exercises not only the file system, but also the CPU and memory system of the test machine.

Because this test showed only a small difference between the version with spinlocks and the version without, and because it corrupted data in the process, we elected not to bias the results of our other benchmarks by attempting them without locking.

The Andrew benchmark reflects file system performance under heavy load, i.e. copying a large number of small files. Because the tests are short, the time is heavily dominated by file creation and deletion. Better performance in those key areas could

description	test 1	test 2	test 3
	test 4	test 5	average
ext2	137.48	137.40	124.18
	132.20	132.21	132.694
linlog	136.59	137.05	137.04
	137.61	138.76	137.204

Table 4.2: Kernel Benchmark.
(times are in seconds)

result in more similar performance between the two file systems in this benchmark.

4.2 Kernel Benchmark

Our second test consisted of a script that copied the linux kernel sources onto our test partition, compiled part of the linux kernel, then deleted the object and source files. This differs from the Andrew Benchmark in that it focuses on read and write performance over an extended period of time.

Times shown in Table 4.2 are total times for the entire process of copying, compiling, deleting, and in the case of LinLogFS, running the cleaner once. In this test, ext2 outperformed LinLogFS by a small margin.

The kernel test represents a more common usage pattern of reading data frequently but writing periodically. We decided to use the cleaner at the end of each cycle, because we feel that this represents a more typical use of the cleaner. Traditionally one would only run the cleaner overnight or even weekly unless the file system is nearly full.

It should be noted that the mean of the LinLog tests falls just within the 95% confidence interval for the mean of the ext2 tests (126.68 to 138.70). These results suggest that the performance of these two file systems is very similar.

4.3 PostMark Benchmark

Because the kernel benchmark results show ext2 being only slightly faster than LinLog, while the Andrew benchmark results show more than a factor of two difference, we elected to add a third benchmark to get a better picture of the differences in performance. Our third test was the PostMark benchmark [7]. The PostMark benchmark is a benchmark that attempts to simulate "heavy small-file system loads", as is typical for many types of server systems (e.g. WWW servers, NNTP servers, mail servers, etc.) It was created by Network Appliance to test performance of their NFS-based file servers.

The PostMark benchmark tests several aspects of performance. During the benchmark, it creates and deletes hundreds of files, measuring the amount of time required to perform those operations. In some cases, it reads the files or appends data to them. When describing creation or deletion, these operations are grouped together and called "transactions". Thus, files that were created and immediately deleted are labeled "creation/deletion alone", while files that were read from or appended to are labeled "mixed w/ transactions".

The results of this test are shown in Table 4.3. This benchmark suite shows

group	test	ext2	linlog
files			
	creation alone	500/sec	500/sec
	mixed w/ transactions	264/sec	264/sec
	read	243/sec	241/sec
	appended	257/sec	256/sec
	deleted	764/sec	750/sec
	deletion alone	528/sec	515/sec
	mixed w/ transactions	236/sec	235/sec
data			
	read	1.36 megs/sec	1.36 megs/sec
	write	4.45 megs/sec	4.38 megs/sec

Table 4.3: PostMark Benchmark.

nearly identical performance in file creation and deletion and in file reads. It shows LinLogFS performing slightly slower on writes (about 2% worse).

4.4 Analysis

The performance of the compile test in the Andrew benchmark is dramatically different from the kernel compile results. One possible reason for this is the length of the test. The Andrew benchmark times were roughly in the sub-10 second range. The kernel compiles, by contrast, were between two and three minutes. Averaging performance over a long time period tends to yield more useful results than a burst test.

The file creation and deletion performance in the PostMark becnmark were significantly different from the results in the Andrew benchmark. One possible cause

is high level caching of file system changes. Because of caching in the Linux VFS layer, a file that is created and immediately deleted may not even reach the file system if this occurs within a sufficiently short period of time. For this reason, the file create and delete speed of both file systems may be exaggerated in PostMark. The Andrew benchmark, by contrast, does not delete files, and is thus largely unaffected by this caching.

Finally, the kernel tests coincide closely with the PostMark file system throughput results. They show that ext2 outperforms LinLogFS by a few percent.

What these numbers fail to account for is the difference in code maturity between LinLogFS and ext2. Clearly, LinLogFS is not as highly tuned as ext2. For example, LinLogFS currently frequently sends debugging messages to the kernel log daemon for display and logging. While we disabled as much of this as we could before benchmarking, the file system remains somewhat noisy.

For comparison purposes, we ran the Andrew benchmark with debug logging fully enabled. The average time for that debugging version was 21.2 seconds, compared to around five seconds for the non-debugging version used for the benchmarks in Section 4.1. Clearly, removing the remaining debug messages is an important step towards making LinLogFS realize its full potential.

Chapter 5

Known Problems

D'oh!
– *Homer.*

Several issues remain in the cleaner. We do not expect these to have a significant impact on performance, but must be addressed before LinLogFS should be considered for use in a production environment. While such issues are frequently glossed over in papers, the nature of this software is such that we felt it necessary to address them directly.

As mentioned previously, the kernel code currently does not attempt to clean files whose inodes are faked, including the .atime file. While the effect of this workaround on performance is expected to be minimal, it is still undesirable.

Under certain circumstances, segment use flags may not be cleared correctly in the file system bitmap. This results in segments getting stuck in a state where they cannot be cleaned. The cause of this problem has yet to be determined.

In a few cases, memory is not freed correctly, or is freed twice. This is caught by the kernel in a relatively safe way, but is still a bug.

Also, there are cases where the memory occupied by the superblock itself may not be freed correctly on unmount. This is believed to be a problem with the file system itself, rather than the cleaner.

Under heavy load, file system corruption can occur if writing to a file is allowed to occur concurrently with cleaning. This is currently prevented by a lock, but this performance degradation should eventually be fixed by correcting whatever issues exist in the current concurrency protection mechanism.

Under stress, the file system may enter a deadlocked state. The reason for this has yet to be determined, but does not appear to be related to the use of the cleaner.

The current number of inodes may not be sufficient for certain uses of the file system, and should be studied more carefully.

When multiple LinLogFS partitions are mounted, the proc file system entries do not work correctly. Unmounting one can cause the other's proc entry to disappear. It is suspected that cleaning a file system while in either state could be hazardous to your data.

Even with debugging disabled, this file system produces a substantial amount of debugging noise. These "error" messages should be re-evaluated, as most are really debugging messages.

Finally, we have adapted the user programs to allow the user to specify a

particular partition. We have also added some sanity checking in argument handling. However our fixes for these programs are only temporary fixes. Ideally, a more flexible solution should be implemented, such as the use of getopt.

There are a number of additional issues specific to the file system itself that are not specific to the cleaner. For more information on these issues, it is suggested that the reader visit the official LinLogFS web site [1].

Chapter 6

Conclusions

> And on the seventh day, He exited from append mode. – *from UNIX fortune*

In conclusion, our tests suggest that the LinLogFS file system performs similarly to ext2, with similar or slightly better sustained read performance, but poorer burst performance, and slightly poorer sustained write performance.

Further, the MrClean cleaner and the kernel code to support it are now largely functional as a result of this effort. The system still contains some bugs, and is thus not recommended for a production environment; however, it is sufficiently usable for performance testing, experimentation, and so on. We feel that this file system has the potential to significantly impact the Linux community in high availability, high performance, and embedded computing (for wear-leveling of flash). We hope that the availability of a working cleaner will foster future development in this area.

LinLogFS and the cleaner are distributed under the terms of the GNU General

Public License (GPL). For information on obtaining our changes via anonymous CVS, visit our web site at http://www.gatwood.net/projects/linlog.html.

Chapter 7

Acknowledgements

Oh, my God! They killed Kenny!
– *South Park.*

I would like to acknowledge a number of people for their contributions without which this work would not have been possible. First and foremost, I would like to thank Christian Czezatke for developing LinLogFS. I would also like to thank Antony Sargent and Trey Cain for creating the cleaner upon which this work was based.

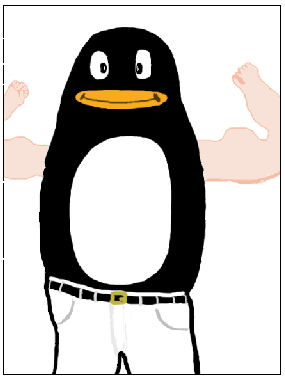

Figure 7.1: Mr. Penguin in the wild.

And I would be remiss if I did not thank my thesis advisor, Tara Madhyastha, for her management without micro-management, her insight, her encouragement, and most importantly, her patience. I'd also like to thank her for not making me take out this sentence.

I would also like to thank my employer, MontaVista Software for allowing flexible hours during the writing of this thesis, and for their continued support of Linux operating system development and enhancement.

I would like to thank the author of LaTeX for making me add a paragraph to my acknolwedgements section to keep from having only a single line on the next page.

I would like to thank Apple Computer for releasing Mac OS X, upon which this was typeset using AppleWorks and, sadly, LaTeX. Also thanks to the Free Software Foundation and specifically the ghostscript team, without whom this would have been more fun than a root canal.

I would also like to thank the UCSC Wind Ensemble for providing a pleasant diversion, and an opportunity to release the daily stress of this thesis through playing 'til my gums bleed.

Special thanks also to Dr. Pepper Company for caffeinating my bloodstream sufficiently to complete this research. Dr. Pepper: for when coffee doesn't work anymore.

I would also like to thank my family, especially my parents, my friends, the Academy, my producer, my director, and all the little people without whom... oops, wrong script.

A special thanks to the (recently departed) Douglas Adams for his wit, particularly in "The Long Dark Tea-Time Of The Soul". The feeling of dread hovering above the buffer pool bugs closely resembled Dirk Gently's feelings towards his worst nightmare: "The fridge no longer merely stood there in the corner of the kitchen, it

actually lurked."

Finally, thank you to everyone else whom I have not mentioned here – even people who have never even heard of this file system – even those who may or may not even be classified as people – in fact, thanks to everything, yes, everything, because in some way, everything either helped or failed to hurt my efforts in this paper, and whether or not that was intentional, or whether or not these things happened strictly by accident or happenstance is the subject of much debate, and yet the question of whether this sentence will end is not, as is the question of whether this paper will end, or rather as is not the question of whether this paper will end, as it most certainly will end... now.

Bibliography

[1] LinLogFS homepage. *http://www.complang.tuwien.ac.at/czezatke/lfs.html*, Accessed 5/18/2001.

[2] Trevor Blackwell. Heuristic cleaning algorithms for log-structured file systems. *Proceedings of the 1995 Usenix Technical Conference*, 1995.

[3] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. *Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9.*

[4] Christian Czezatke. dtfs – a log-structured file system for Linux. *http://www.complang.tuwien.ac.at/czezatke/linlogpapers.html*, 1998.

[5] J. Howard. An overview of the andrew file system. *USENIX Winter Technical Conference:23–26, February 1988.*

[6] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems, 6(1):51–81, February 1988.*

[7] Jeffery Katcher. Postmark: A new file system benchmark. *Technical Report TR-3022, Network Appliance.*

[8] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. *Proceedings of the 1995 USENIX Technical Conference, Jan. 1995:155–164.*

[9] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems, 2(3):81–197, August 1984.*

[10] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). *Proceedings of ACM SIGMOD, Chicago, IL, June, 1988, pp. 109 – 116.*

[11] Antony Sargent and Trey Cain. MrClean – implementing a cleaner for the Linux log-structured file system. *http://www.cs.wisc.edu/ cain/mrclean.ps*, 1998.

[12] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. The design and implementation of the 4.4BSD log-structured file system. *Proceedings of the 1993 Winter Usenix, January 1993, San Diego, CA*.

[13] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. A log-structured file system for UNIX. *Proceedings of the 1993 Winter Usenix Conference*, 1993.

[14] Vinod Valloppillil. "Halloween document", open source software: A (new?) development methodology. *http://www.opensource.org/halloween/halloween1.html*.