File System Performance and Transaction Support

by

Margo Ilene Seltzer

A.B. (Harvard/Radcliffe College) 1983

A dissertation submitted in partial satisfaction of the

requirements of the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Michael Stonebraker, Chair
Professor John Ousterhout
Professor Arie Segev

1992

File System Performance and Transaction Support

by

Margo Ilene Seltzer

Abstract

File System Performance and Transaction Support

by

Margo Ilene Seltzer

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Michael Stonebraker, Chair

This thesis considers two related issues: the impact of disk layout on file system throughput and the integration of transaction support in file systems.

Historic file system designs have optimized for reading, as read throughput was the I/O performance bottleneck. Since increasing main-memory cache sizes effectively reduce disk read traffic [BAKER91], disk write performance has become the I/O performance bottleneck [OUST89]. This thesis presents both simulation and implementation analysis of the performance of read-optimized and write-optimized file systems.

An example of a file system with a disk layout optimized for writing is a log-structured file system, where writes are bundled and written sequentially. Empirical evidence in [ROSE90], [ROSE91], and [ROSE92] indicates that a log-structured file system provides superior write performance and equivalent read performance to traditional file systems. This thesis analyzes and evaluates the log-structured file system presented in [ROSE91], isolating some of the critical issues in its design. Additionally, a modified design addressing these issues is presented and evaluated.

Log-structured file systems also offer the potential for superior integration of transaction processing into the system. Because log-structured file systems use logging techniques to store files, incorporating transaction mechanisms into the file system is a natural extension. This thesis presents the design, implementation, and analysis of both user-level transaction management on read and write optimized file systems and embedded transaction management in a write optimized file system.

This thesis shows that both log-structured file systems and simple, read-optimized file systems can attain nearly 100% of the disk bandwidth when I/Os are large or sequential. The improved write performance of LFS discussed in [ROSE92] is only attainable when garbage collection overhead is small, and in nearly all of the workloads examined, performance of LFS is comparable to that of a read-optimized file system. On transaction processing workloads where a steady stream of small, random I/Os are issued, garbage collection reduces LFS throughput by 35% to 40%.

Dedication

To Nathan Goodman
for believing in me when I doubted myself, and for helping me find large mountains and move them.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I have been fortunate to have had many brilliant and helpful influences at Berkeley. My advisor, Michael Stonebraker, has been patient and supportive throughout my stay at Berkeley. He challenged my far-fetched ideas, encouraged me to pursue whatever caught my fancy, and gave me the freedom to make my own discoveries and mistakes. John Ousterhout was a member of my thesis and qualifying exam committees. His insight into software systems has been particularly educating for me and his high standards of excellence have been a source of inspiration. His thorough reading of this dissertation improved its quality immensely. Arie Segev was also on my qualifying exam and thesis committees and offered sound advice and criticism.

The interactions with Professors Dave Patterson and Randy Katz rounded out my experience at Berkeley. They have discovered how to make computer science into ‘‘big science’’ and to create enthusiasm in all their endeavors. I hope I can do them justice by carrying this trend forward to other environments.

I have also been blessed with a set of terrific colleagues. Among them are my co-authors: Peter Chen, Ozan Yigit, Michael Olson, Mary Baker, Etienne Deprit, Satoshi Asami, Keith Bostic, Kirk McKusick, and Carl Staelin. The Computer Science Research Group provided me with expert guidance, criticism, and advice, contributing immensely to my technical maturation. I owe a special thanks to Kirk McKusick who gave up many hours of his time and his test machine to make BSD-LFS a reality. Thanks also go to the the Sprite group of Mary Baker, John Hartman, Mendel Rosenblum, Ken Shirriff, Mike Kupfer, and Bob Bruce who managed to develop and support an operating system while doing their own research as well! They were a constant source of information and assistance.

Terry Lessard-Smith and Bob Miller saved the day on many an occasion. It seemed that no matter what I needed, they were always there, willing to help out. Kathryn Crabtree has also been a savior on many an occasion. It has always seemed to me that her job is to be able to answer all questions, and I don’t think she ever let me down. Th transition to graduate school would have been impossible without her help and reassuring words. Those who claim that graduate school is cold and impersonal didn’t spend enough time with people like Kathryn, Bob, and Terry.

There are many other people who have offered me guidance and support over the past several years and they deserve my unreserved thanks. My officemates, the inhabitants of Sin City: Anant Jhingran, Sunita Sarawagi, and especially Mark Sullivan, have been constant sources of brain power, entertainment, and support. Mike Olson, another Sin City inhabitant, saved the day on many papers and my dissertation by making troff sing. Mary Baker, of the Sprite project, has been a valued colleague, devoted friend, expert party planner, chef extraordinairre, and exceptionally rigorous co-author. If I can hold myself to the high standards Mary sets for herself, I am assured a successful career.

Then there are the people who make life just a little more pleasant. Lisa Yamonaco has known me longer than nearly anyone else and continues to put up with me and offer unconditional love and support. She has always been there to share in my successes and failures, offer words of encouragement, provide a vote of confidence, or just to make me smile. I am grateful for her continued friendship.

Ann Almgren, my weekly lunch companion, shared many of my trials and tribulations both in work and in play. Eric Allman was always there when I needed him to answer a troff question, fix my sendmail config files, provide a shoulder, or invite me to dinner. His presence made Berkeley a much more pleasant experience. Sam Leffler was quick to supply me with access to

Silicon Graphics' equipment and source code when I needed it, although I've yet to finish the research we both intended for me to do! He has also been a devoted soccer fan and and a good source of diversions from work. My friends and colleagues at Quantum Consulting were always a source of fun and support.

Life at Berkeley would have been dramatically different without the greatest soccer team in the world, the Berkeley Bruisers, particularly Cathy Corvello, Kerstin Pfann, Brenda Baker, Robin Packel, Yvonne Gindt, and co-founder Nancy Geimer. They've kept my body as active as my mind and helped me maintain perspective during this crazy graduate school endeavor. A special thanks goes to Jim Broshar for over four years of expert coaching. More than teaching soccer skills, he helped us craft a vision and discover who we were and who we wanted to become.

Even with all my support in Berkeley, I could never have survived the last several years without my electronic support network, the readership of *MISinformation*. The occasional pieces of email and reminders that there was life outside of graduate school helped to keep me sane. I look forward to their continued presence via my electronic mailbox.

And finally, I would like to thank Keith Bostic, my most demanding critic and my strongest ally. His technical expertise improved the quality of my research, and his love and support improved the quality of my life.

This research has been funded by the National Science Foundation grants NSF-87-15235 and IRI-9107455, the National Aeronautics and Space Administration grant NAG-2-530, the Defense Advanced Research Projects Agency grants DAALO3-87-K-0083 and DABT63-92-C-0007, and the California State Micro Program.

Chapter 1

Introduction

As CPU speeds have increased dramatically over the past decade, I/O performance is becoming more and more of a system bottleneck [PATT88]. Therefore, improving system throughput has become the task of the designers of I/O subsystems and file systems. While I/O subsystem designers improve the hardware with disk arrays, faster busses, and larger caches, software designers can try to use the existing systems more efficiently. This thesis addresses how file systems can be modified to use existing I/O systems more efficiently.

Maximum disk performance can be achieved by reading and writing the disk sequentially, avoiding costly disk seeks. The traditional wisdom has been that data is read far more often than it is written, and therefore, files should be allocated sequentially on disk so that they can be read sequentially. However, today's large main memory caches effectively reduce disk read traffic, but do little to reduce write traffic [OUST89]. Anticipating the growing importance of write performance on I/O performance and overall system performance, a great deal of file system research is focused on improving write performance.

Evidence suggests that as systems become faster and disks and memories become larger, the need to write data quickly will also increase. The file system trace data in [BAKER91] demonstrates that in the past decade, files have become larger. At the same time, CPUs have become dramatically faster and high-speed networks have enabled applications to move large quantities of data very rapidly. These factors make it increasingly important that file systems be able to move data to and from the disk quickly.

File system performance is normally tied to the intended application workload. In the workstation and time-sharing markets, where files are read and written in their entirety, the Berkeley Fast File System (FFS) [MCKU84], with its rotation optimization and logical clustering, has been relatively satisfactory. In the database and super-computing worlds, the tendency has been to choose file systems that favor the contiguous disk layout offered by extent-based systems. However, when the workload is diverse, including both of these application types, neither file system is entirely satisfactory. In some cases, demanding applications such as database management systems manage their own disk allocation. This results in static partitioning of the available disk space and maintaining two or more separate sets of utilities to copy, rename, or remove files. If the initial allocation of disk space is incorrect, the result is poor performance, wasted space or both. A file system that offers improved performance across a wide variety of workloads would simplify system administration and serve the needs of the user community better.

This thesis examines existing file systems, searching for one that provides good performance across a wide range of workloads. The file system design space can be divided into read-optimized and write-optimized systems. Read-optimized systems allocate disk space contiguously to optimize for sequential accesses. Write-optimized systems use logging to optimize writing large quantities of data. One goal of this research is to characterize how these different strategies respond to different workloads and use this characterization to design better performing file systems.

This thesis also examines using the logging of a write-optimized file system to integrate transaction support with the file system. This embedded support is compared to traditional user-level transaction support. A second goal of this research is to analyze the benefit of integrating transaction support in the file system.

Chapter 2 presents previous work related to this dissertation. It begins with a discussion of how file systems have used disk allocation policies to improve performance. Next, several alternative transaction processing implementations are presented. The chapter concludes with a summary of some evaluations of file systems and transaction processing systems.

Chapter 3 presents a simulation study of several read-optimized file system designs. The simulation uses three stochastically generated workloads that model time-sharing, transaction processing, and super-computing workloads to measure read-optimized file systems that use multiple block sizes. The file systems are evaluated based on effective disk utilization (how much of the total disk bandwidth the file systems can use), internal fragmentation (the amount of allocated but unused space), and external fragmentation (the amount of unallocated, but usable space on the disk).

Chapter 4 focuses on the transaction processing workload. It presents a simulation study that compares read-optimized and write-optimized file systems for supporting transaction processing. It also contrasts the performance of user-level transaction management with operating system transaction management. The specific write-optimized file system analyzed is the log-structured file system first suggested in [OUST89]. This chapter shows that a log-structured file system has some characteristics that make it particularly attractive for transaction processing.

Chapter 5 presents an empirical study of an implementation of transaction support embedded in a log-structured file system. This implementation is compared to a conventional user-level transaction implementation. This chapter identifies several important issues in the design of log-structured file systems.

Chapter 6 presents a new log-structured file system design based on the results of Chapter 5.

Chapter 7 presents the performance evaluation of the log-structured file system design in Chapter 6. The file system is compared to a the Fast File System and an extent-based file system on a wide range of benchmarks. The benchmarks are based upon database, software development, and super-computer workloads.

Chapter 8 summarizes the conclusions of this work.

Chapter 2

Related Work

This chapter discusses several classes of research, related to this dissertation. As this thesis presents an evaluation of file system allocation policies and transaction processing support, there are three main categories of related work: file systems, transaction systems, and evaluations. The file system sections discuss a number of different allocation policies and how the state of the art has evolved over time. The transaction processing section presents several alternative implementation strategies for providing transaction support to the user. Some of these different strategies will be analyzed in Chapters 4 and 5 of this dissertation. The evaluation section summarizes five studies that analyze transaction processing performance.

2.1. File Systems

The file systems are sub-divided into two classes: read-optimized and write-optimized file systems. Read-optimized systems assume that data is read more often than it is written and that performance is maximized when files are allocated contiguously on disk. Write-optimized file systems focus on improving write performance, sometimes at the expense of read performance. This division of allocation policies will be used throughout this work to describe different file systems. The examples presented here provide an historical background to the evolution of file system allocation strategies.

2.1.1. Read-Optimized File Systems

Read-optimized systems focus on sequential disk layout and allocation, attempting to place files contiguously on disk to minimize the time required to read a file sequentially. Simple systems that allocate fixed-sized blocks can lead to files becoming fragmented, requiring repositioning the disk head for each block read, leading to poor performance when blocks are small. Attempting to allocate files contiguously on disk reduces the head movement and improves performance, but requires more sophisticated bookkeeping and free space management. The six systems described present a range of alternatives.

2.1.1.1. IBM's Extent Based File System

IBM's MVS system provides *extent-based* allocation. An extent is a unit of contiguous on-disk storage, and files are composed of some number of extents. When a user creates a new file, she specifies a *primary extent size* and a *secondary extent size*. The primary extent size defines how much disk space is initially allocated for the file while the secondary extent size defines the size of additional allocations [IBM]. If users know how large their files will become, they can select appropriate extent sizes, and most files can be stored in a few large contiguous extents. In such cases, these files can be read and written sequentially and there is little wasted space on the disk. However, if the user does not know how large the file will grow, then it is extremely difficult to select extent sizes. If the extents are too small, then performance will suffer, and if they are too large, there will be a great deal of wasted space. In addition, managing free space and finding extents of suitable size becomes increasingly complex as free space becomes more and more fragmented. Frequently, background disk rearrangers must be run during off-peak hours to coalesce free blocks.

2.1.1.2. The UNIX[1] V7 File System

Systems with a single block size (*fixed-block systems*), such as the original UNIX V7 file system [THOM78] solve the problems of keeping allocation simple and fragmentation to a minimum, but they do so at the expense of efficient read and write performance. In this file system, files are composed of some number of 512-byte blocks. An unsorted list of free blocks is maintained and new blocks are allocated from this list. Unfortunately, over time, as many files are created, rewritten, and deleted, logically sequential blocks within a file are scattered across the entire disk, and the file system requires a disk seek to retrieve each block. Since each block is only 512 bytes, the cost of the seek is not amortized over a large transfer. Increasing the block size reduces the per-byte cost, but it does so at the expense of internal fragmentation, the amount of space that is allocated but unused. As most files are small [OUST85], they fit in a single, small block. The unused, but allocated space in a larger block is wasted. Sorting the free list allows small blocks to be accessed more efficiently by allocating them in such a way as to avoid a disk seek between each access. However, this necessitates traversing half of the free list, on average, for every deallocation.

2.1.1.3. The UNIX Fast File System

The BSD Fast File System (FFS) [MCKU84] is an evolutionary step forward from the simple fixed-block system. Files are composed of a number of fixed-sized *blocks* and a few smaller *fragments*. Small fragments alleviate the problem of internal fragmentation described in the previous system. The larger blocks, on the order of 8 or 16 kilobytes, provide for more efficient disk utilization as more data is transferred per seek. Additionally, the free list is maintained as a bit map so that blocks may be allocated in a rotationally optimal fashion without spending a great deal of time traversing a free list. The rotational optimization makes it possible to retrieve successive blocks of the same file during a single rotation, thus reducing the disk access time. File allocation is clustered so that logically related files, those created in the same directory, are placed on the same or a nearby cylinder to minimize seeks when they are accessed together.

2.1.1.4. Extent-like Performance on the Fast File System

McVoy suggests improvements to the Fast File System in [MCVO91]. He uses block clustering to achieve performance close to that of an extent-based system. The FFS block allocator remains unchanged, but the *maxcontig* parameter, which defines how many blocks can be placed contiguously on disk, is set equal to 64 kilobytes divided by the block size. The 64 kilobytes, called the cluster size, was chosen not to exceed the maximum transfer allowed on any controller.

When the file system translates logical block numbers into physical disk requests, it determines how many logically sequential blocks are contiguous on disk. Using this number, the file system can read more than one logical block in a single I/O operation. In order to write clusters, blocks that have been modified (dirty blocks) are cached in memory and then written in a single I/O. By clustering these relatively small blocks into 64 kilobyte clusters, the file system achieves performance nearly identical to that of an extent-based system, without performing complicated allocation or suffering severe internal fragmentation.

2.1.1.5. The Dartmouth Time Sharing System

In an attempt to merge the fixed-block and extent-based policies, the DTSS system described in [KOCH87] is a file system that uses binary buddy allocation [KNUT69]. Files are composed of extents, each of whose size is a power of two (measured in sectors). Files double in size whenever their size exceeds their current allocation. Periodically (once every day in DTSS), a reallocation algorithm runs. This reallocator changes allocations to reduce both the internal and external fragmentation. After reallocation, most files are allocated in 3 extents and average under 4%

internal fragmentation. While this system provides good performance, the reallocator necessitates quiescing the system each evening which is impractical in many environments.

2.1.1.6. Restricted Buddy Allocation

The restricted buddy system is a file system with multiple block sizes, initially described and simulated in [SELT91], that does not require a reallocator. Instead of doubling allocations and fixing them later as in DTSS, a file's block size increases gradually as the file grows. Small files are allocated from small blocks, and therefore do not suffer excessive internal fragmentation. Large files are mostly composed of larger blocks, and therefore offer adequate sequential performance. Simulation results discussed in [SELT91] and Chapter 3, show that these systems offer performance comparable to extent-based systems and small internal fragmentation comparable to fixed-block systems. Restricted buddy allocation systems do not require reorganization, avoiding the down time that DTSS requires.

2.1.2. Write-Optimized File Systems

Write-optimized systems focus on improving the performance of writes to the file system. Because large, main-memory file caches more effectively reduce the number of disk reads than disk writes, disk write performance is becoming the system bottleneck [OUST89]. The trace driven analysis in [BAKER91] shows that client workstation caches reduce application read traffic by 60%, but only reduce write traffic by 10%. As write performance begins to dominate I/O performance, write-optimized systems will become more important.

The following systems focus on providing better write performance rather than improving disk allocation policies. The first two systems described in this section, DECorum and The Database Cache, have disk layouts similar to those described in the read-optimized systems. They improve write performance by logging operations before they are written to the actual file system. The second two systems, Log Files and The Log-structured File System, change the on-disk layout dramatically, so that data can be written directly to the file system efficiently.

2.1.2.1. DECorum

The DECorum file system [KAZ90] is an enhancement to the Fast File System. When FFS creates a file or allocates a new block, several different on-disk data structures are updated (block bit maps, inode bit maps, and the inode). In order to keep all these structures consistent and expedite recovery, FFS performs may operations (file creation, deletion, rename, etc) synchronously. These synchronous writes penalize the system in two ways. First, they increase latency as operations wait for the writes to complete. Secondly, they result in additional I/Os since data that is frequently accessed may be repeatedly written. For example, each time a file is created or deleted, the directory containing that file is synchronously written to disk. If many files in the same directory are created/deleted, many additional I/Os are issued. These additional I/Os can take up a large fraction of the disk bandwidth.

The DECorum file system uses a write-ahead logging technique to improve the performance of operations that are synchronous in the Fast File System. Rather than performing synchronous operations, DECorum maintains a log of the modifications that would be synchronous in FFS. Since FFS semantics allow the system to lose up to 30 seconds worth of updates [MCKU84], and DECorum is supporting the same semantics, the log need only be flushed to disk every 30 seconds. As a result, DECorum avoids many I/Os entirely, by not repeatedly writing indirect blocks as new blocks are appended to the file and by never writing files which are deleted within the 30 second window. In addition, all writes, including those for inodes and indirect blocks, are asynchronous. Write performance, particularly appending to the end of a file, improves. Read performance remains largely unchanged, but since the file system is performing fewer total I/O's,

---

UNIX is a trademark of Unix System Laboratories.

overall disk utilization should decrease leading to better read response time. In addition, the logging improves recovery time, because the file system can be restored to a logically consistent state by reading the log and aborting or undoing any partially completed operations.

2.1.2.2. The Database Cache

The database cache, described in [ELKH84], extends the idea in DECorum one step further. Instead of logging only meta-data operations in memory, the database cache technique improves write performance by logging dirty pages sequentially to a large cache, typically on disk. The dirty pages are then written back to the conventional file system asynchronously to make room in the cache for new pages. On a lightly loaded system, this will improve I/O performance because most writes will occur at sequential speeds and blocks accumulate in the cache slowly enough that they may be sorted and written to the actual file system efficiently. However, in some applications such as those found in an online transaction processing environment this writing from the cache to the database can still limit performance. At best, the database cache technique will sort I/O's before issuing writes from the cache to the disk, but simulation results show that even well-ordered writes are unlikely to achieve utilization beyond 40% of the disk bandwidth [SELT90].

2.1.2.3. Clio's Log Files

The V system's [CHER88] Clio logging service extends the use of logging to replace the file system entirely [FIN87]. Rather than keep a separate operation log or database cache, this file system is designed for write-once media and is represented as a readable, append-only log. Files are logically represented as a sequence of records in this log, called a sublog. The physical implementation gathers a number of log records from one or more files to form a block. In order to access a file, index information, called an *entry map*, is stored in the log. Every N blocks, a level 1 entry map is written. The level 1 entry map contains a bit map for each file found in the preceding N blocks, indicating in which blocks the file has log records. In order to find particular records within a block, the block is scanned sequentially. Every N^2 blocks a level 2 entry map is written. Level 2 entry maps contain per-file bit maps indicating in which level 1 entry map the files appear. In general, level i entry maps are written every N^i blocks and indicate in which level $i-1$ entry maps a particular file can be found. Figure 2-1 depicts this structure, where $N = 4$.

Entry maps can grow to be quite large. In the worst case, where every file is composed of one record, entry maps require an entry for every file represented. If records of the same file are scattered across many blocks, then many blocks are sequentially scanned to find the file's records. As a result, while the Clio system provides good write performance as well as logging and history capabilities, the read performance is hindered by the hierarchical entry maps and sequential scanning within each map and block.

2.1.2.4. The Log-structured File System

The Log-Structured File System, as originally proposed by Ousterhout in [OUST89], provides another example of a write-optimized file system. As in Clio, a log-structured file system (LFS) uses a log as the only on-disk representation of the file system. Files are represented by an inode that contains the disk addresses of data blocks and indirect blocks. Indirect blocks contain disk addresses of other blocks providing an index tree structure to access the blocks of a file. In order to locate a file's inode, a log-structured file system keeps an inode map which contains the disk address of every file's inode. This structure is shown in Figure 2-2.

Both LFS and Clio can accumulate a large amount of data and write it to disk sequentially, providing good write performance. However, the LFS indexing structure is much more efficient than Clio's entry maps. Files are composed of blocks so there is no sequential scanning within blocks to find records. Furthermore, once a file's inode is located, at most three disk accesses are

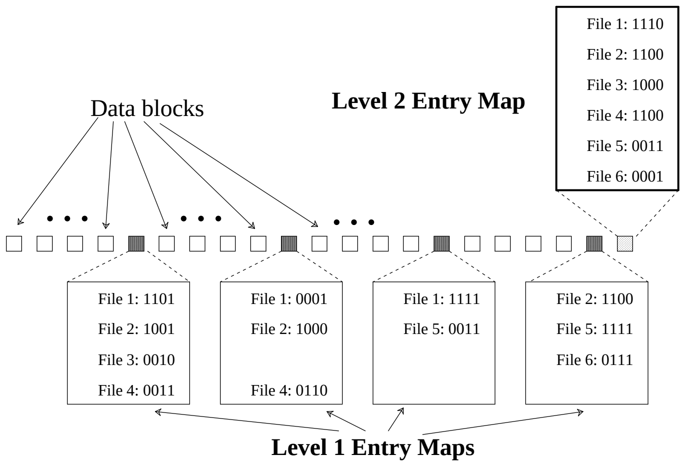


Figure 2-1: Clio Log File Structure. This diagram depicts a log file structure with N=4. Each data block contains a sequence of log records. The entry maps indicate which blocks contains records for which files. For example, the level 2 entry map indicates that file 1 has blocks in the first three level 1 entry maps, but file 3 has blocks only in the first level 1 entry map. Since the bit map for file 1 in the first level 1 entry map contains the value ‘‘1101’’, file 1 has records located in the first, second, and fourth blocks described by that entry map. It also has records in the fourth block described by the second level 1 entry map and all the blocks described by the third level 1 entry map.

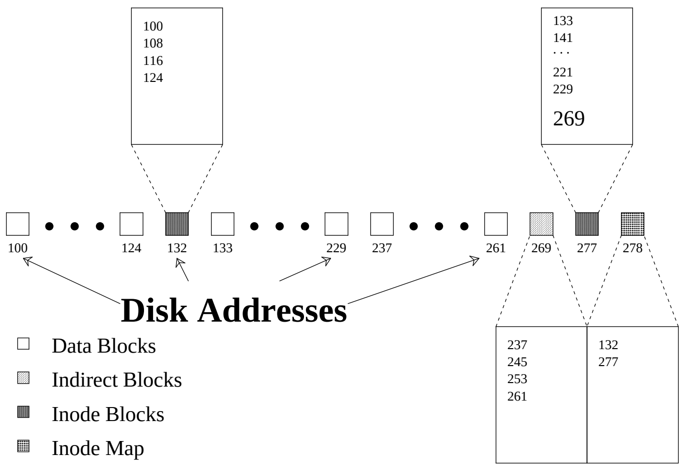


Figure 2-2: Log-Structured File System Disk Allocation. This diagram depicts the on-disk representation of files in a log-structured file system. In this diagram, two inode blocks are shown. The first contains blocks that reside at disk addresses 100, 108, 116, and 124. The second contains many direct blocks, allocated sequentially from disk address 133 through 268, and an indirect block, located at address 269. While the inode contains references to the data blocks from disk address 133 through 236, the indirect block references the remainder of the data blocks. The last block shown is part of the inode map and contains the disk address of each of the two inodes.

required to find any particular item in the file, regardless of the file system size. In contrast, the number of disk accesses required in Clio grows as the number of allocated blocks increases. While Clio keeps all records to provide historical retrieval, LFS uses a garbage collector to reclaim space from files that have been modified or deleted. Therefore an LFS file system is usually more compact (as space is reclaimed), but the cleaner competes with normal file system activity for the disk arm.

2.2. Transaction Processing Systems

The next set of related work discusses transaction systems. Although the goal of this thesis is to find a file system design which performs well on a variety of workloads, the transaction processing workload is examined most closely. In particular, two fundamentally different transaction architectures are discussed. In the first, *user-level*, transaction semantics are provided entirely as user-level services, while in the second, *embedded*, transaction services are provided by the operating system.

The advantage of user-level systems is that they usually require no special operating system support and may be run on different platforms. Although not a requirement of the user-level architecture, these systems are typically offered only as services of a database management system (DBMS). As a result, only those applications that use the DBMS can use transactions. This is a disadvantage in terms of flexibility, but can be exploited to provide performance advantages. When the data manager is the only user of transaction services, the transaction system can use semantic information provided by database applications. For example, locking and logging operations may be performed at a logical, rather than physical, granularity. This usually means that less data is logged and a higher degree of concurrency is sustained.

There are three main disadvantages of user-level systems. First, as discussed above, they are often only available to applications of the DBMS and are therefore, somewhat inflexible. Second, they usually compete with the operating system for resources. For example, both the transaction manager and the operating system buffer recently-used pages. As a result, they often both cache the same pages, using twice as much memory. Third, since transaction systems must be able to recover to a consistent state after a crash, user-level systems must implement their own recovery paradigm. The operating system must also recover its file systems, so it too implements a recovery paradigm. This means that there are multiple recovery paradigms. Unfortunately, recovery code is notoriously complex and is often the subsystem responsible for the largest number of system failures [SULL91]. Supporting two separate recovery paradigms is likely to reduce system availability.

The advantages of embedded systems are that they provide a single system recovery paradigm, and they typically offer a general purpose mechanism available to all applications, not just the clients of the DBMS. There are two main disadvantages of these systems. First, since they are embedded in the operating system, they usually have less detailed knowledge of the data and cannot perform logical locking and logging. This can result in performance penalties. Second, if the transaction system interferes with non-transaction applications, overall system performance suffers.

The next two sections introduce each architecture in more detail and discuss systems representing the architecture.

2.2.1. User-Level Transaction Support

This section considers several alternatives for providing transaction support at user-level. The most common example of these systems are the commercial database management systems discussed in the next section. Since commercial database vendors sell systems on a variety of different platforms and cannot modify the operating systems on which they run, they implement all transaction processing support in user-level processes. Only DBMS applications, such as database servers, interactive query processors and programs linked with the vendor's application libraries, can take advantage of transaction support. Some research systems, such as ARGUS [LISK83] and Eden [PU86], provide transactions through programming language support, but in this section, only the more general mechanisms that do not require new or modified languages are considered.

2.2.1.1. Commercial Database Management Systems

Oracle and Sybase represent two of the major players in the commercial DBMS market. Both companies market their software to end-users on a wide range of platforms, and they both provide a user-level solution for data management and transaction processing. In order to provide good performance, Sybase takes exclusive control of some part of a physical device, which it then uses for extent-based allocation of database files [SYB90]. The Sybase SQL Server provides hierarchical locking for concurrency control and logical logging for recovery. Oracle has a similar architecture. It can either take control of a physical device or allocate files in the file system. Oracle also takes advantage of the knowledge that only database applications will be using the concurrency control and recovery mechanisms, so it performs locking and logging on logical units as well [ORA89]. This is the architecture used for user-level transaction management in this thesis.

2.2.1.2. Tuxedo

The Tuxedo system from AT&T is a transaction manager which coordinates distributed transaction commit across heterogeneous local transaction managers. While it provides support for distributed two-phase commit, it does not actually include its own native transaction mechanism. Instead, it could be used in conjunction with any of either the user-level or embedded transaction systems described here or in [ANDR89].

2.2.1.3. Camelot

Camelot's distributed transaction processing system [SPE88A] provides a set of Mach [ACCE86] processes which provide support for nested transaction management, locking, recoverable storage allocation, and system configuration. In this way, most of the mechanisms required to support transaction semantics are implemented at user-level, but the resulting system can be used by any application, not just clients of a data manager.

Applications can make guarantees of atomicity by using Camelot's recoverable storage, but requests to read and write such storage are not implicitly locked. Therefore, applications must make requests of the disk manager to provide concurrency control [SPE88B]. The advantage of this approach is that any application can use transactions, but the disadvantage is that such applications must make explicit lock requests to do so.

2.2.2. Embedded Transaction Support

The systems described in the next section provide examples of the ways in which transactions have been incorporated into operating systems. Computer manufacturers like IBM, Tandem, Stratus, and Hewlett-Packard include transaction support directly in the operating system. The systems described present a range of alternatives. The first three systems, Tandem's ENCOMPASS, Stratus TPF, and Hewlett-Packard's MPE, provide general purpose operating system transaction mechanisms, available to any applications. In these systems, specific files are identified as being transaction protected and whenever they are accessed, appropriate locking and logging is performed. These systems are most similar to those discussed in Chapters 3 and 4.

The next system, LOCUS, uses atomic files to make the distributed system recoverable. This is similar to Camelot's recoverable storage, but is used as the system-wide data recovery mechanism. The last system, Quicksilver, takes a broader perspective, using transactions as the single recovery paradigm for the entire system.

2.2.2.1. Tandem's ENCOMPASS

Tandem Computers manufactures a line of fault tolerant computers called NonStop Systems[2], designed expressly for online transaction processing [BART81]. Guardian 90 is their message-based, distributed operating system which provides services required for high performance online transaction processing [BORR90]. Although this is an embedded system, it was designed to provide all the flexibility that user-level systems provide. Locking is performed by processes that manage the disks (disk servers) and allows for hierarchical locking on records, keys, or fragments (parts of a file) with varying degrees of consistency (browse, stable reads, and repeatable reads). In order to provide recoverability in the presence of fine-grain locking, Guardian performs *logical UNDO logging* and *physical REDO logging*. This means that a logical description of the operation (e.g. field one's value of 10 was overwritten) is recorded to facilitate aborting a transaction, and the complete physical image of the modified page is recorded to facilitate recovery after a crash. Application designers use the Transaction Monitoring Facility (TMF) application interface to build client/server applications which take advantage of the concurrency control and recovery present in the Guardian operating system [HELL89].

2.2.2.2. Stratus' Transaction Processing Facility

Stratus Computer offers both embedded and user-level transaction support [STRA89]. They support a number of commercial database packages which use user-level transaction management, but also provide an operating system based transaction management facility to protect files not managed by any DBMS. This is a very general purpose mechanism that allows a file to be transaction-protected by issuing the *set_transaction_file* command. Once a file has been designated as transaction protected, it can only be accessed within the context of a transaction. It may be opened or closed outside a transaction, but attempts to read and write the file when there is no active transaction in progress will result in an error.

Locking may be performed at the key, record, or file granularities. Each file has an *implicit* locking granularity which is the size of the object that will be locked by the operating system in the absence of explicit lock requests by the application. For example, if a file has an implicit key locking granularity, then every key accessed will be locked by the operating system, unless the application has already issued larger granularity locks. In addition, a special end-of-file locking mode is provided to allow concurrent transactions to append to files.

Transactions may span machines. A daemon process, the TPOverseer, implements two-phase commit across distributed machines. At each site, the local TPOverseer uses both a log and shadow paging technique [ASTR76]. During phase 1 commit processing (the preparation phase), the application waits while the log is written to disk. When a site completes phase 1, it has guaranteed that it is able to commit the transaction. During phase 2 (the actual commit), the shadow pages are incorporated into the actual files.

This model is similar to the operating system model simulated in Chapter 4 and implemented in Chapter 5. However, when this architecture is implemented in a log-structured file system, the logging and shadow paging are part of normal file system operation as opposed to being additional independent mechanisms.

2.2.2.3. Hewlett-Packard's MPE System

Hewlett-Packard integrates operating system transactions with their memory management and physical I/O system. Transaction semantics are provided by means of a memory-mapped write-ahead log. Those files which require transaction protection are marked as such and may then be accessed in one of two ways. First, applications can open them for mapped access, in which case the file is mapped into memory and the application is returned a pointer to the beginning of the

[2] NonStop and TMF are trademarks of Tandem Computers.

file. Hardware page protection is used to trigger lock acquisition and logging on a per-page basis. Alternatively, protected files can be accessed via the data manager. In this case, the data manager maps the files and performs logical locking and logging based on the data requested [KOND92]. This system demonstrates the tightest integration between the operating system, hardware, and transaction management. The advantage of this integration is very high performance at the expense of transaction management mechanisms permeating nearly every part of the MPE system.

2.2.2.4. LOCUS

The LOCUS distributed operating system [WALK83] provides nested, embedded transactions [MUEL83]. There are two levels to the implementation. The basic LOCUS operating system uses a shadow page technique to support atomic file updates on all files. On top of this atomic file facility, LOCUS implements distributed transactions which use a two-phase commit protocol across sites. Locks are obtained both explicitly, by system calls, and implicitly by accessing data. While applications may explicitly issue unlock requests, the transaction system retains any locks that must be held to preserve transaction semantics. The basic atomic file semantics of LOCUS are similar to the LFS embedded transaction manager that will be discussed in Chapter 5, except that in LOCUS, the atomic guarantees are enforced on all files rather than on those optionally designated. If LFS enforced atomicity on all its files, it could also be used as the basis for a distributed transaction environment.

2.2.2.5. Quicksilver

Quicksilver is a distributed system which uses transactions as its intrinsic recovery mechanism [HASK88]. Rather than providing transactions as a service to the user, Quicksilver, itself, uses transactions as its single system-wide architecture for recovery. In addition to providing recoverability of data, transaction protection is applied to processes, window management, network interaction, etc. Every interprocess communication in the system is identified with a transaction identifier. Applications can make use of Quicksilver's built-in services by adding transaction identifiers to any IPC message to associate the message and the data accessed by that message with a particular transaction. The Quicksilver Log Manager provides a low-level, general purpose interface that makes it suitable for different servers or applications to implement their own recovery paradigms [SCHM91]. This is the most pervasive of the transaction mechanisms discussed. While it is attractive to use a single recovery paradigm (e.g. transactions) this thesis will focus on isolating transaction support to the file system.

2.3. Transaction System Evaluations

This section summarizes several evaluation studies that include file system transaction support, operating system transaction systems, and operating system support for database management systems. The first study compares two transactional file systems. The functionality provided by these systems is similar to the functionality provided by the file system transaction manager described in Chapter 5. The second, third, and fourth evaluations discuss the difficulties in providing operating system mechanisms for transaction processing and data management. The last evaluation presents a simulation study that compares user-level transaction support to operating system transaction support. This study is very similar to the one presented in Chapter 4.

2.3.1. Comparison of XDFS and CFS

The study in [MITC82] compares the Xerox Distributed File System (XDFS) and the Cambridge File System (CFS), both of which provide transaction support as part of the file system. CFS provides atomic objects, allowing atomic operations on the basic file system types such as files and indices. XDFS provides more general purpose transactions, using stable storage to make guarantees of atomicity. The analysis concludes that XDFS was a simpler system, but provided slower performance than CFS, and that CFS' single object transaction semantics were too restrictive. This thesis will explore an embedded transaction implementation with the potential for providing the simplicity of XDFS with the performance of CFS.

2.3.2. Operating System Support for Databases

In [STON81], Stonebraker discusses the inadequate support for databases found in the operating systems of the day. His complaints fall into three categories: a costly process structure, slow and suboptimal buffer management, and small, inefficient file system allocation, Fortunately, much has changed since 1981 and many of these problems have been addressed. Operating system threads [ANDE91] and lightweight processes [ARAL89] address the process structure issue. Buffer management may be addressed by having a data base system manage a pool of memory-mapped pages so that the data manager can control replacement policies, perform read-ahead, and access pages as quickly as it can access main memory while still sharing memory equitably with the operating system. This thesis will consider file system allocation policies which improve allocation efficiency.

2.3.3. Virtual Memory Management for Database Systems

Since the days of Multics [BEN69], memory mapping of files has been suggested as a way to reduce the complexity of managing files. Even so, database management systems tend to provide their own buffer management. In [TRA82], Traiger looks at two database systems, System R [ASTR76] and IMS [IBM80] and shows that memory mapped files do not obviate the need for database buffer management. Although System R and IMS use different mechanisms for transaction support (shadow-paging and write-ahead logging respectively), neither is particularly well suited to the use of memory mapped files.

Traiger assumes that a mapped file's blocks are written to paging store when they are evicted from memory. However, today's systems, such as those designs in [ACCE86] and [MCKU86], treat mapped files as memory objects which are backed by files. Thus, when unmodified pages are evicted from memory, they need not be written to disk, because they can later be reread from their backing file. Additionally, modified pages can be written directly to the backing file, rather than to paging store.

There are still difficulties in using memory-mapped files for databases and transactions. Consider the write-ahead logging protocol of IMS. If the virtual memory system is responsible for writing back pages, the transaction system needs some mechanism to guarantee that log records are written to disk before their associated data pages. Similar problems are encountered in shadow-paging. The page manager must be able to change memory-mappings to remap shadow pages. The 1982 study correctly concludes that memory-mapped files do not obviate the need for additional database or transaction buffer management.

2.3.4. Operating System Transactions for Databases

The criticisms of operating system transactions continue with [STON85] which reports on experiences in trying to port the INGRES database management system [RTI83] on top of Prime's Recovery Oriented Access Method (ROAM) [DUBO82]. ROAM is designed to provide atomic updates to files with all locking, logging, and recovery hidden from the user. However, when INGRES was ported to this mechanism, several problems were encountered. First, a single record update in INGRES modifies two sets of bytes on a page, the line table and the record itself. In order for ROAM to properly handle this, it either had to log entire pages or perform two separate log operations, both costly alternatives. Secondly, since ROAM did page level locking, updates to system catalogs had extremely detrimental effects on the level of concurrency, as a single modification to a catalog would lock out all other users. One approach to improving the concurrency on the system catalogs is to allow short term locking. However, short term locking makes recoverability more complicated since concurrent transactions may access the data modified by an uncommitted transaction. Stonebraker concludes by suggesting the following alternatives: allowing user processes to log events, designing database systems so that only physical events need to be rolled back, and leaving everything at user-level as traditional data managers do today. The next study discusses the performance ramifications of the second alternative.

2.3.5. User-Level Data Managers v.s. Embedded Transaction Support

Kumar concludes that an operating system embedded transaction manager provides substantially worse performance than the traditional user-level data manager [KUM87]. He cites the inability to perform logical locking and logging, the system call locking overhead, and the size of the log as primary causes for a 30% difference in performance between the two systems. In [KUM89], by introducing hardware-assisted locking and better locking protocols, he demonstrates that the difference in performance may be reduced to 7-10%. However, Kumar's simulation failed to account for the required write when evicting dirty buffers from the cache. Since these are random I/O's, his results under-report the total I/O time. Specifically, in disk-bound configurations, performance is dominated by the cost of random I/O's. Since both the data manager and embedded systems perform the same number of these random reads and writes, performance should be virtually the same in both models, not dependent upon the log writes which happen at sequential disk speeds.

2.4. Conclusions

The work in this dissertation will touch upon all the different areas discussed in this section. Chapter 3 focuses on the read-optimized allocation policies. Chapter 4 presents a study similar to Kumar's, adding the simulation of a log-structured file system. Chapter 5 analyzes the tradeoffs between user-level and embedded transaction systems with an implementation study. Chapter 6 presents a new design for a log-structured file system, and Chapter 7 analyzes the differences in application performance of read-optimized and write-optimized file systems.

Chapter 3

Read-Optimized File Systems

This chapter compares several read-optimized file system allocation policies. These file systems are designed to provide high bandwidth between disks and main memory by taking advantage of parallelism in an underlying disk array and catering to large units of transfer. In this way, when files are written in their entirety as they are in most UNIX environments [OUST85] [BAKER91], these file systems may be competitive with write-optimized file systems. The goal of these read-optimized designs is to utilize as much of the I/O bandwidth as possible when reading sequentially, without sacrificing small-file efficiency in terms of disk capacity. Typically, small blocks are preferred to minimize fragmentation for small files, and large blocks or contiguous allocation is preferred to maximize throughput for large files.

In this chapter, the read-optimized file systems are divided into two categories: fixed-block systems and extent-based systems. Fixed-block systems allocate files as collections of identically sized blocks while extent-based systems allocate files as collections of a few large extents whose sizes may vary from file to file. Traditionally, systems oriented towards general-purpose timesharing (e.g. UNIX) have used fixed-block systems, while systems oriented towards transaction processing (e.g. MVS) have chosen extent-based systems. Fixed-block file systems have received much criticism from the database community. The most frequently cited criticisms are discontiguous allocation and excessive amounts of meta-data [STON81]. On the other hand, extent-based file systems are often criticized for being too brittle with regard to fragmentation and too complicated in terms of allocation.

In Chapter 2, many styles of read-optimized file systems were discussed. The simulation presented here focuses on three of the extent or multiple-block-sized systems and one fixed-block system. The multiple-block-sized systems analyzed are an extent-based system similar to IBM's MVS system, a binary buddy system similar to DTSS [KOCH87], and a restricted buddy system. The fourth system is a fixed-block system similar to FFS, but without fragments.

The goal of this chapter is to analyze how well different allocation policies perform without the use of an external reallocation process. The file systems are compared in terms of fragmentation and disk system throughput. The rest of this chapter is organized as follows. Section 3.1 presents the simulation model and Section 3.2 establishes the evaluation criteria used throughout the rest of the chapter. Section 3.3 introduces the different allocation policies and the simulation results that characterize each, and Section 3.4 compares the policies against one another.

3.1. The Simulation Model

The four allocation policies are analyzed by means of an event driven, stochastic workload simulator. There are three primary components to the simulation model: the disk system, the workload characterization, and the allocation policies. The disk system and workload characterization are described in Sections 3.1.1 and 3.1.2, while the allocation policies are described in detail in Section 3.3.

3.1.1. The Disk System

The disk system is an array of disks, viewed as a single logical disk. Although many such systems use additional parity drives to improve system availability, for simplicity, the simulated array does not contain parity. Blocks are numbered so that data written in large, contiguous units to the logical disk will be striped across the physical disks. When data is striped across disks, there are two parameters which characterize the layout of disk blocks, the *stripe unit* and the *disk unit*. The stripe unit is the number of bytes allocated on a disk before allocation is performed on the next disk. This unit must be greater than or equal to the sector sizes of all the disks. The disk unit is the minimum unit of transfer between a disk and memory. This is the smaller of the smallest block size supported by the file system and the stripe unit. Disk blocks are addressed in terms of disk units.

Each disk is described in terms of its physical layout (track size, number of cylinders, number of platters) and its performance characteristics (rotational speed and seek parameters). The seek performance is described by two parameters, the one track seek time and the incremental seek time for each additional track. If ST is the single track seek time and SI is the incremental seek time, then an N track seek takes $ST + N*SI$ ms. Table 3-1 contains a listing of the parameters which describe one common disk and its default values for these simulations.

3.1.2. Workload Characterization

The workload is characterized in terms of file types and their reference patterns, similar to the synthetic trace generator described in [WRI91]. A simulation configuration consists of any number of file types, defined by their size characteristics, access patterns, and growth characteristics. Table 3-2 summarizes those parameters which define a file type.

For each file type, initialization consists of two phases. In the first phase, *nusers* events are created, and each is assigned a start time uniformly distributed in the range [0, (*nusers* * *hfreq*)], where *hfreq* is the average time between requests from different users. The events are maintained sorted in scheduled time order. During the second initialization phase, the files are created. The initial file sizes are selected from a normal distribution with mean *i_size* and deviation *i_dev*. Allocation requests are issued for each file until the file has reached its initial size.

Disk Parameters For the CDC 5 ¼" Wren IV Drives (94171-344)		
	actual	simulated
N Disks	NA	8
Total Capacity	2.8 G	2.8 G
Max Throughput	10.8 M/sec	10.8 M/sec
N Platters	9	9
N Cylinders	1549	1600
Bytes/Track	24 K	24 K
1 Track Seek Time	5.5 ms	5.5 ms
Inc. Seek Time	0.0320 ms	0.0320 ms
Rotational Latency	16.67 ms	16.67

Table 3-1: Disk Parameters and Default Values

File Parameters	
nfiles	Number of files created
nusers	Number of parallel events
ptime	Milliseconds between requests from a single user
hfreq	Milliseconds between requests from different users
rw_size	Mean size of each read/write operation
rw_dev	Standard deviation in read/write size
a_size	For extent-based systems, mean extent size
t_size	Mean size of deallocate requests
i_size	Mean initial file size
i_dev	Deviation in the mean file size.
r_ratio	Percent read operations.
w_ratio	Percent write operations.
e_ratio	Percent extend operations.
d_ratio	Percent deallocates which are file deletes.
t_ratio	Percent deallocates which are truncates.

Table 3-2: File Parameters and Description

The simulation runs by selecting the next scheduled event, determining the type of event (allocation, deallocation, read or write), processing that event, and scheduling a new request. Events correspond to file types. For each file type, the read ratio (*r_ratio*), write ratio (*w_ratio*), truncate ratio (*t_ratio*), extend ratio (*e_ratio*), and delete ratio (*d_ratio*) indicate what percent of the requests are of the particular type. The size of an allocation, read, or write operation is selected from a normal distribution with mean *rw_size* and deviation *rw_dev*. The size of a truncation operation is also drawn from a normal distribution, but with a mean of *t_size*. After the operation is completed, an exponentially distributed value with mean equal to *ptime* is added to the time at which the operation completed, and an event is scheduled at that newly calculated time. If an allocation request cannot be satisfied, a disk full condition is logged, and the current event is rescheduled (exponentially distributed with mean *ptime*).

There are two types of simulations: allocation tests and throughput tests. Allocation tests are used to determine the fragmentation in the file system and throughput tests report bandwidth utilization. The two metrics are measured separately since allocation tests require filling the disk to capacity while throughput tests need to run until the throughput has stabilized, and disk full conditions would distort the measured throughput. Allocation tests are terminated the first time that an allocation request fails. Throughput tests are terminated by one of two conditions, either a specified number of milliseconds have been simulated or the throughput of the system has stabilized. The system is assumed to stabilize when two conditions have been met: three successive short-term measurements (throughput for a ten-second period) are the same and the short-term measurement is equal to the long-term measurement (throughput for the entire simulation duration). Typically, the simulations stabilized within 24 simulated hours.

Three workloads are used to simulate a time-sharing or software development environment (TS), a large transaction processing environment (TP), and a super-computer or complex query processing environment (SC).

The time-sharing workload is based loosely on the trace driven analyses in [OUST85] and [BAKER91] and is characterized by an abundance of small files (mean size 8 kilobytes) which are created, read, and deleted. If a file is deleted, the next request to read that file will first create it. Therefore, the workload which creates, reads, and deletes files is composed of 50% reads and 50% deletes with the creates caused implicitly. Five-sixths of all requests are to these small files, while the remaining one-sixth are to larger files (mean size 96 kilobyte). The large files are usually read (60% of the time) and occasionally extended, written or truncated (15% writes, 15% extends, 5% deletes and 5% truncates).

The transaction processing workload is based loosely on the TP/1 [ANON85] and TPC-B benchmarks [TPCB90]. It is characterized by eight large files (210 megabytes each) representing data files or relations, five small application logs (5 megabytes each), and one transaction log (10 megabytes). The relations are read and written randomly (60% reads, 30% writes), and infrequently extended and truncated (7% extends, 3% truncates). It is assumed that log files are never deleted and that the abort rate is relatively low, so that log files are rarely read. The system log receives a slightly higher read percentage to simulate transaction aborts.

The super-computer workload is based on the trace study presented in [MILL91]. The environment is characterized by 1 large file (500 megabytes), 15 medium sized files (100 megabytes each), and 10 small files (10 megabytes each). The large file and seven of the medium files are all read and written in large, contiguous bursts (.5 megabyte) with a predominance of reads (60% reads, 30% writes, 8% extends, and 2% truncates). The rest of the medium files and the small files are read and written in 8 kilobyte bursts, but are periodically deleted and recreated (60% reads, 30% writes, 5% extends, 5% deletes). Table 3-3 summarizes the different workloads.

3.2. Evaluation Criteria

The two evaluation criteria for each policy are disk utilization and throughput. The metrics for measuring disk utilization are the *external fragmentation* (amount of space available when a request cannot be satisfied) and *internal fragmentation* (the fraction of allocated space that does not contain data). The allocation tests are run by performing only the extend, truncate, delete, and create operations in the proportion expressed by the file type parameters. As soon as the first allocation request fails, the external and internal fragmentation are computed.

The metrics for throughput are expressed as a percent of the sustained sequential performance of the disk system. For example, the configuration shown in Table 3-1 is capable of providing a sustained throughput of 10.8 megabytes/sec. Therefore, a throughput of 1.1 megabytes/sec is expressed as 10% of the maximum available capacity.

Throughput is calculated for two sets of tests, the application performance test and the sequential performance test. For the application performance test, the application workloads described in the previous section are applied. For the sequential test, files are read and written in their entirety. Thus, the sequential test gives an upper bound on the performance provided by the disk system for a particular allocation policy.

3.3. The Allocation Policies

This section describes the four file systems simulated, including a discussion of the selection of the relevant parameters for each model. The first file system is a binary buddy system similar to that described in [KOCH87]. Files are composed of a fixed number of extents, each of whose size is a power of two (measured in sectors). Files double in size when they exceed their current allocation. The next file system is a restricted buddy system which supports only a few different block sizes. The third is the extent-based policy described in [STON89]. The fourth system is a simple, fixed-block system. It uses rotational positioning and clustering like the FFS, but uses only a single block size (i.e. it does not support fragments).

Workload	Characteristic	File Type 1	File Type 2	File Type 3	File Type 4
Time Sharing	type	small	medium		
	access pattern	whole file	whole file		
	% requests	87.0	13.0		
	num files	25000	12000		
	mean size	8 KB	96 KB		
	% reads	50.0	60.0		
	% writes	0.0	15.0		
	% truncates	0.0	5.0		
	% extends	0.0	15.0		
	% deletes	50.0	5.0		
Transaction Processing	type	data file	application log	transaction log	
	access pattern	random	append	append	
	run length	(4 KB)	(256 bytes)	(128 bytes)	
	% requests	62.5	12.5	25.0	
	num files	8	5	1	
	mean size	210 MB	5 MB	10 MB	
	% reads	60.0	2.0	5.0	
	% writes	30.0	0.0	0.0	
	% truncates	3.0	5.0	1.0	
	% extends	7.0	93.0	94.0	
	% deletes	0.0	0.0	0.0	
Super Computer	type	large	medium	medium	small
	access pattern	sequential	sequential	random	random
	run length	(.5 MB)	(.5 MB)	(8 KB)	(8 KB)
	% requests	4.0	30.0	33.0	33.0
	num files	1	7	8	10
	mean size	500 MB	100 MB	100 MB	10 MB
	% reads	60.0	60.0	60.0	60.0
	% writes	30.0	30.0	30.0	30.0
	% truncates	2.0	2.0	2.0	0.0
	% extends	8.0	8.0	8.0	5.0
	% deletes	0.0	0.0	0.0	5.0

Table 3-3: Workload Characterizations

3.3.1. Binary Buddy Allocation

The binary buddy allocation policy described in [KOCH87] includes both an allocation process and a background reallocation process that runs during off-peak hours. This simulation considers only the allocation and deallocation algorithm (i.e. not the background reallocation). This will not impact performance, as the performance benefit is derived from the large extents, and reallocation only runs when the file system is not being used. However, the resulting fragmentation numbers will be exaggerated relative to what they would be after relocation.

In the buddy allocation system, a file is composed of some number of extents. The size of each extent is a power of two multiple of the sector size. Each time a new extent is required, the extent size is chosen to double the current size of the file. Figure 3-1 depicts this allocation policy.

As previous work suggests [KNOW65] [KNUT69], such policies are prone to severe internal fragmentation, and the simulation results, in Table 3-4, confirm this. However, since there are a small number of extents, very high throughput is observed when large files are present. The throughput results in Table 3-4 show that when large files are present, as in the super-computer and transaction processing workloads, sequential access uses over 93% of the total bandwidth. Since most of the accesses are quite large in the super-computer workload, even the application tests are able to utilize 88% of the available throughput. When files are small, as in the time-sharing environment, or when many accesses are random (as in transaction processing) the resulting throughput is much lower. Therefore, this policy works extremely well for workloads which demand large, sequential accesses, but does little to improve random or small file performance.

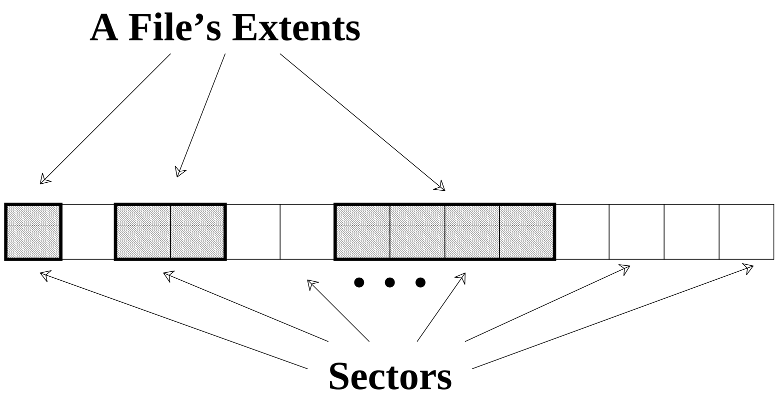


Figure 3-1: Allocation for the Binary Buddy Policy. Files are allocated by doubling their extent size whenever the file exceeds the current allocation. The disk can initially be thought of as one large block. When an allocation of size N is required, and no such free block currently exists, the smallest block of size greater than or equal to N is divided into two equal blocks. The process repeats until a block of the appropriate size is created.

Workload	Disk Usage		Throughput	
	Internal Fragmentation (% allocated space)	External Fragmentation (% total space)	Application Performance (% max throughput)	Sequential Performance (% max throughput)
SC	43.1%	13.4%	88.0%	94.4%
TP	15.2%	9.0%	27.7%	93.9%
TS	18.4%	2.3%	8.4%	12.0%

Table 3-4: Fragmentation and Performance Results for Buddy Allocation.

3.3.2. Restricted Buddy System

As in the binary buddy system, the restricted buddy system uses the principle that a file's unit of allocation should grow as the file's size grows. Additionally, logically sequential allocations within a file are placed contiguously whenever possible. Therefore, when successive allocations are placed contiguously on disk, multiple allocation units can be transferred in a single I/O. To improve small file performance by reducing both the number and length of seeks, the disk is divided into regions to allow clustering of blocks within a file when they cannot be allocated sequentially.

The potential difficulties of such a system are threefold. Supporting multiple allocation sizes makes maintaining free space and allocating disk space complex and could increase external fragmentation. Growing block sizes may increase internal fragmentation for files using only part of a large block. It may be difficult to provide good performance for small files since the cost of a seek between two logically sequential blocks will be amortized across very little data, and will therefore make small-file performance poor.

Each of these problems can be addressed by limiting the complexity of the design. External fragmentation is addressed by restricting the number of allocation sizes, allocating disk blocks in a manner that favors keeping large contiguous regions unused, and selecting block sizes which are multiples of each other. Minimizing internal fragmentation is addressed by carefully selecting the point at which the block size grows. Efficient access for small files is provided by taking advantage of the underlying disk structure. In a single disk system, that means placing blocks in rotationally optimal positions on the same cylinder so that multiple blocks may be retrieved in a single rotation. On a multi-disk system, this means numbering blocks so that requests to sequential blocks can be serviced by multiple disks in parallel.

3.3.2.1. Maintaining Contiguous Free Space

Keeping track of free space can become complex when maintaining blocks of various sizes. The two major questions to be answered are at what granularity free space should be recorded (the largest blocks or the smallest blocks), and what data structure should be used. If free space is maintained only in terms of maximum-sized blocks then a separate mechanism is required to allocate small blocks within the larger blocks. If free space is maintained by bit maps in terms of minimum sized blocks, then it becomes difficult to find larger sized blocks. One would have to search the bit map for a potentially large number of contiguous free blocks. Furthermore, it is difficult to maintain large contiguous free areas when servicing small allocations. These issues have led to the adoption of a hierarchical free space strategy.

The disk system is divided into regions called bookkeeping regions. A bookkeeping region is roughly analogous to the Fast File System's cylinder groups and is described by a *bookkeeper*. Each bookkeeping region has its own free space structures. It maintains a bit map that describes the entire region in terms of maximum-sized blocks. When a smaller block is required, a large block is allocated, its bit it toggled, and it is broken up into blocks of the next smaller size. Free block information for these smaller blocks is maintained in a sorted, doubly-linked list. Within each of these lists, the bookkeeper points to the next block to be allocated. Each disk unit of the region is represented in the bit map (in terms of its associated maximum-sized block) and also in one free list if it is unallocated. In this way, blocks of various sizes can be found quickly.

3.3.2.2. File System Parameterization

A restricted buddy file system may be parameterized to suit a particular environment. The three main sets of parameters are: the block sizes, the bookkeeping unit size, and the grow policy.

The size of a bookkeeping region is the unit of clustering within the file system. It must be at least as large as twice the largest block size and is usually much larger. If files are expected to grow to be quite large, larger bookkeeping regions are desirable so that a large number of maximum-sized blocks are available. However, if most files are not expected to require maximum-sized blocks, smaller bookkeeping regions will realize more benefits of tight clustering. Since allocation becomes more difficult as the disk fills, at the time of file system creation, one may specify how much of the file system space should be left free.

The grow policy determines when the size of the allocation unit is increased and is expressed in terms of a multiplier. If g is the grow policy multiplier and the block sizes are a_i, then the unit of allocation increases from a_i to a_{i+1} when the sum of the sizes of all blocks of size a_i is equal to $g * a_{i+1}$. For example, a system with block sizes of 1 kilobyte and 8 kilobytes and a grow policy multiplier (grow factor) of 1 will allocate eight 1-kilobyte blocks before allocating any 8-kilobyte blocks. If the next larger block size were 64 kilobytes, then eight 8-kilobyte blocks would be allocated before growing the block size to 64 kilobytes. Intuitively, one expects that a smaller grow factor will cause worse internal fragmentation (since bigger blocks are used in smaller files), but might offer better performance (since fewer small block transfers are required). However, if the small blocks are allocated contiguously, then the performance should be comparable, and the larger grow factor is desirable.

3.3.2.3. Allocation and Deallocation

When the allocation manager receives an allocation request, it attempts to satisfy that request from the optimal bookkeeping region. The goal in selecting regions and blocks is similar to that of the FFS, in that it attempts to select a block that is conveniently close to associated blocks. Additionally, it must try to maintain large contiguous regions of unallocated space for large block allocation requests. The definition of the optimal region depends on the type of request. If the request is for a block of a file, the optimal region is that region that contains the most recently allocated block for that file. If no blocks have been allocated, the optimal region is that region in which the file's index structure (inode) was allocated. If the allocation request is for an inode, the optimal region is the region containing the inode's parent directory. Finally, if the request is for an inode, but that inode represents a directory (during directory creation), the inode is allocated to that region containing the lowest *free split ratio*. The free split ratio is the ratio of the amount of free space that cannot be used for maximum-sized blocks divided by the amount of free space represented in contiguous maximum-sized blocks. If two regions have the same free split ratio, the region with the greater amount of free space is selected. This balances three conflicting goals: clustering related data, spreading new directories, and maintaining maximum-sized blocks.

If a request is made to a specific region, and there is adequate contiguous space, but no block of the appropriate size, then a larger block is split. The larger block is removed from its free list or bit map. A block of the desired size is allocated, and the remaining space is linked into the free lists for the smaller blocks. If the request fails in the desired region, it is passed up to the free split block algorithm which looks for a region with a free block of the appropriate size. If no blocks of the appropriate size are found in any region, only then is a larger block split. Once a split becomes necessary, the region with the best free split ratio is selected, unless the desired allocation is for the largest sized block, in which case, the block with the lowest free split ratio is selected. Table 3-5 summarizes the total allocation strategy.

When a block is deallocated, it is reattached onto the appropriate free list of the appropriate bookkeeping region. Entries on free lists are maintained in sorted order so that coalescing may be performed at deallocation time. Any block which is not of the smallest block size in the file system is called a parent block, and is composed of N child blocks (blocks of the next smaller allocation size). When block B is deallocated, if B's remaining sibling blocks are unallocated and present in the free list, then all N child blocks (B and its siblings) are coalesced and removed from their free list, and the parent of B is added to its free list. In this way, blocks on the free list will always be of the greatest possible size. Using this coalescing algorithm, the number of entries in these free lists is expected to be quite low, as observed in the DTSS binary block

Select Optimal Region
- same as last
- same as inode
- same as directory
- max free split ratio

Select region with a free block of the correct size and the greatest free split ratio
Select the region with the greatest free split ratio
Select region with the most free space

Table 3-5: Allocation Region Selection Algorithm.

system [KOCH87]. In practice, the average list length was under 4 (3.63).

3.3.2.4. Exploiting the Underlying Disk System

In order to provide good performance in the presence of many small files, the file system needs to use the underlying disk system efficiently, avoiding extraneous seeks on a single disk system and exploiting parallelism on a multiple disk system. The Fast File System (FFS) and the Log-structured File System (LFS) both provide effective mechanisms for optimizing the single disk case, so this section will consider how to best exploit the parallelism in a multi-disk configuration by spreading data across multiple disks. Simply speaking, to optimize for large files, large blocks are automatically striped across the disks, and to optimize for small files, different files are explicitly scattered across the disks.

The disk system is addressed as a linear address space of disk units. Each block size is an integral multiple of the disk unit and of all the smaller block sizes. In order to keep allocation simple, a block of size N always starts at an address which is an integral multiple of N. If a system supports block sizes of 1 kilobyte and 8 kilobytes, the 1-kilobyte blocks located at addresses 0 through 7 are considered buddies, together forming a block of size 8 kilobytes. Whenever possible, buddies are allocated sequentially to the same file and are coalesced at deallocation.

The parameters that define a file system in the restricted buddy policy are the number of block sizes, the specific sizes, when to increase the block size (the grow policy), and whether or not to attempt to cluster allocations for the same file. Four different sets of block sizes, two different algorithms for choosing when to increase the block size, and both clustered and unclustered policies are considered. The four block size configurations are:

Number of Block Sizes	Block Sizes
2	1K, 8K
3	1K, 8K, 64K
4	1K, 8K, 64K, 1M
5	1K, 8K, 64K, 1M, 16M

In order to group allocations within a file physically close to one another, the allocator attempts to placed allocations of the same file in the same bookkeeping region. For each set of block sizes, both a clustered configuration, with 32 megabyte bookkeeping regions, and an unclustered configuration were considered.

The allocation and throughput tests were run on all the workloads described in Section 3.1.2. Figure 3-2 shows the fragmentation results. The most striking result is that the attempt to

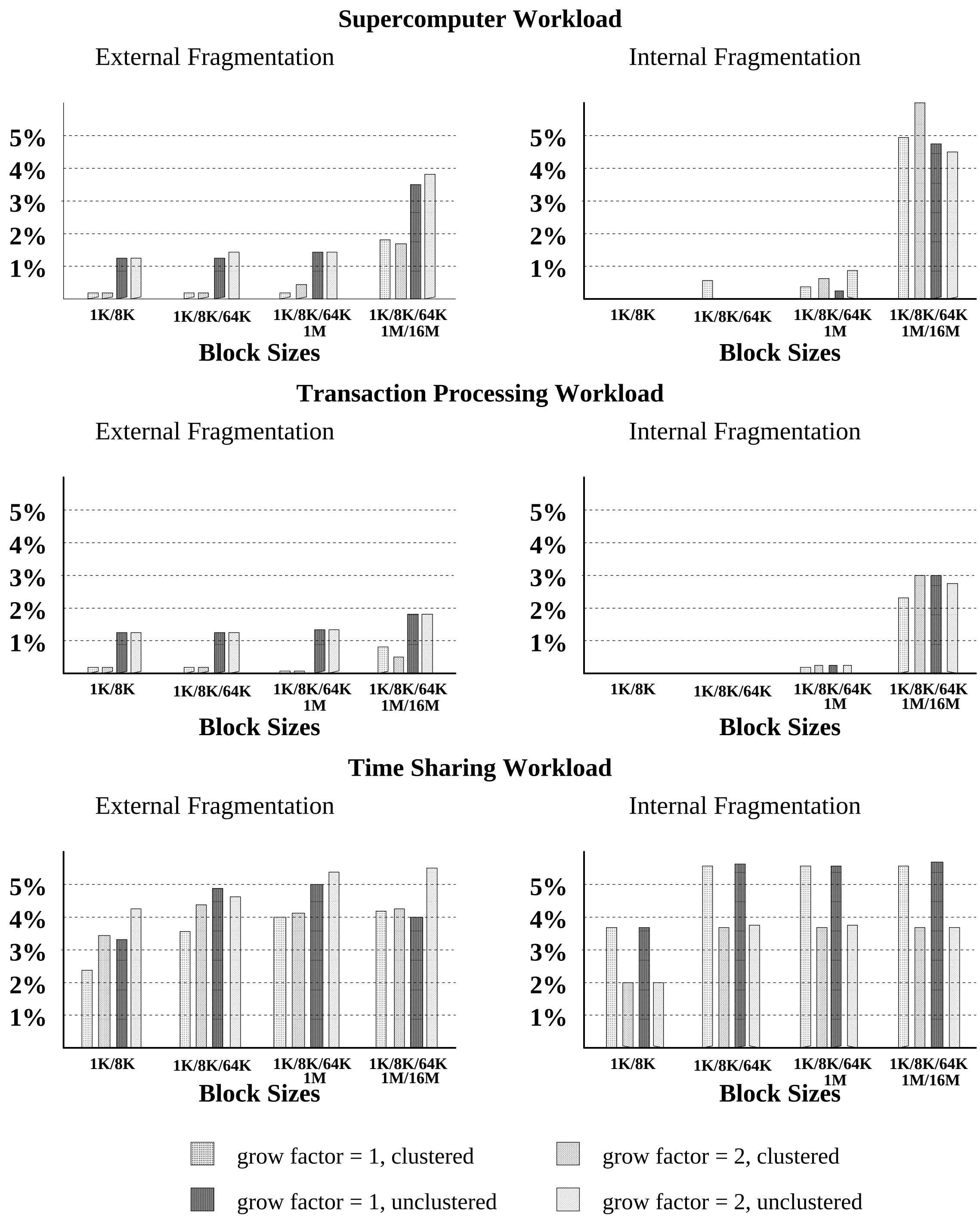


Figure 3-2: Fragmentation for the Restricted Buddy Policy. Each pair of graphs shows the internal and external fragmentation for the indicated workload. None of the policies produce either internal or external fragmentation in excess of 6%.

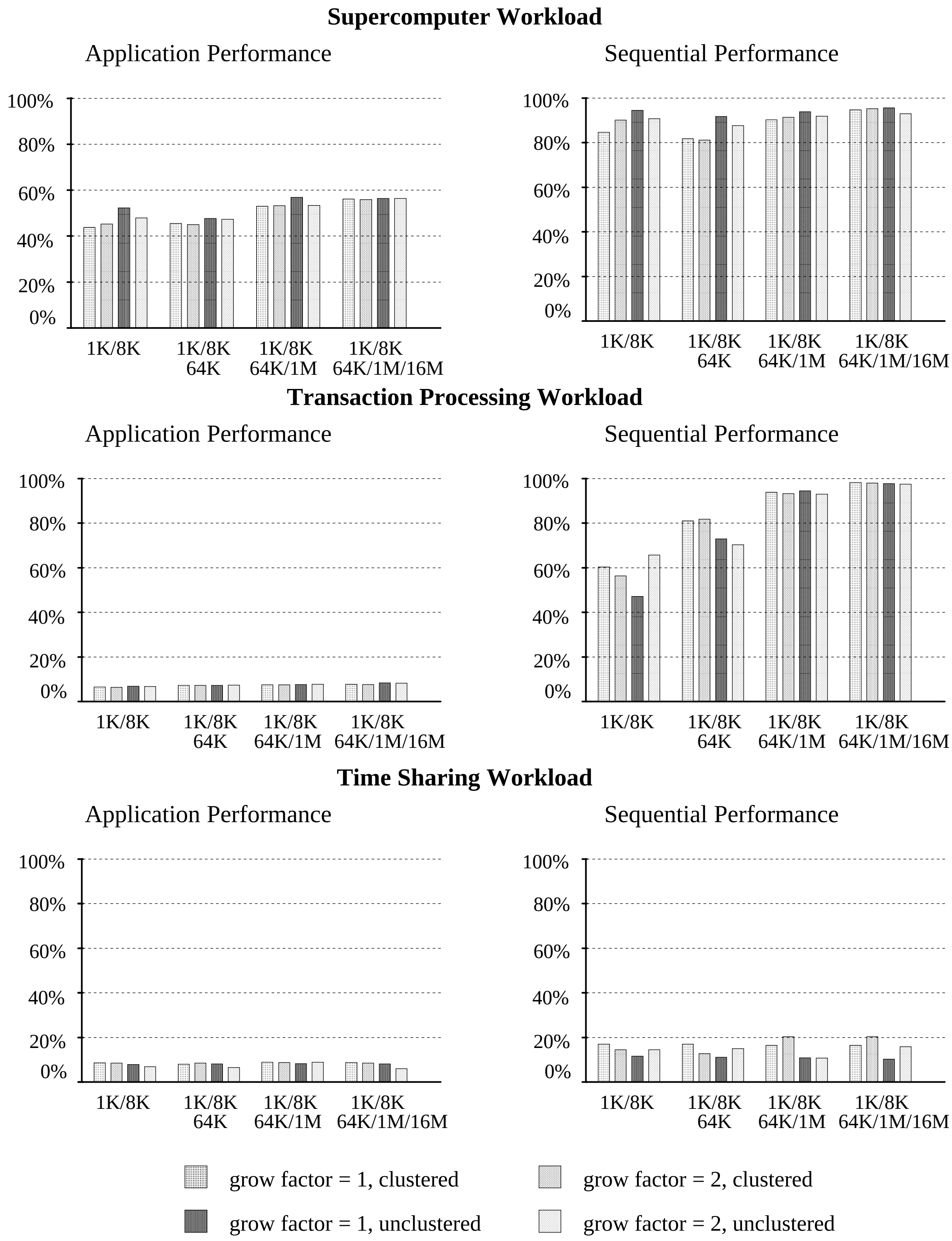


Figure 3-3: Application and Sequential Performance for the Restricted Buddy Policy.

coalesce free space and maintain large regions for contiguous allocation is successful. None of the polices produce either internal or external fragmentation greater than 6%. Part of the explanation for this lies in the static file population in the simulation. Since the ratio of large to small files remains constant, small files continue to be allocated from small blocks, and the large blocks remain available for large files. Still, the time-sharing workload, which has the blend of large and small files, exhibits the greatest fragmentation, and fragmentation increases as the number of blocks sizes and the block sizes themselves increase. Increasing the grow factor from one to two reduces the internal fragmentation by approximately one-third (the difference between each pair of adjacent bars in the upper right-hand graph). External fragmentation increases slightly in an unclustered configuration since a larger selection of blocks is eligible for splitting (all blocks in the disk system instead of just those in a specific region).

Figure 3-3 shows the results of the application and sequential tests for the three workloads under each configuration of the restricted buddy policy. As expected, the configurations which support the larger block sizes provide the best throughput, particularly where large files are present (the top four graphs in Figure 3-3). The super-computer application in the first two graphs shows up to 25% improvement for configurations with large blocks, while the transaction processing environment shows an improvement of 20%. These same workloads are relatively insensitive to either the grow policy or clustering. For the five-block-size configuration (the rightmost on each graph), most show slightly better performance with an unclustered configuration. The explanation of this phenomena lies in the movement of files between regions. In a clustered configuration, when a change of region is forced, the location of the next block is random with regard to the previous allocation. In an unclustered configuration, there are typically only small seeks between subsequent allocations and the performance is slightly better.

The time-sharing workload reflects the greatest sensitivity to the clustering and grow policy. Uniformly, clustering tends to aid performance, by as much as 20% in the sequential case (in the lower right-hand graph of Figure 3-3, the first two bars of each set represent the clustered configuration and the third and fourth bars represent the unclustered configuration). Since this environment is characterized by a greater number of smaller files, data is being read from disk in fairly small blocks even with the larger block sizes. As a result, the seek time has a greater impact on performance, and the clustering policy which reduces seek time provides better throughput.

The graph on the bottom right indicates that the higher grow factor provides better throughput (the second and fourth bars in each set represent a grow factor of two, while the first and third bars represent a grow factor of one). This is counter-intuitive since a higher grow factor means that more small blocks are allocated. To understand this phenomena, one needs to analyze how the attempt to allocate blocks sequentially interacts with the grow policy. Figure 3-4 shows a 1-megabyte block that is subdivided into sixteen 64-kilobyte blocks, each of which is subdivided into eight 8-kilobyte blocks. When the grow factor is one, any file over 72 kilobytes requires a 64-kilobyte block. However, when it is time to acquire a 64-kilobyte block, the next sequential 64-kilobyte block is not contiguous to the blocks already allocated. In contrast, when the grow factor is two, the 64-kilobyte block isn't required until the file is already 144 kilobytes. Since most files in the timesharing workload are smaller than this, they never pay the penalty of performing the seek to retrieve the 64-kilobyte block. Thus our grow policy and our attempts to lay out blocks contiguously are in conflict with one another, and the grow policy should be modified to allow contiguity between different sized blocks.

Using the results of this section, a configuration for comparison with the other allocation policies was selected. Since the larger blocks sizes did not increase fragmentation significantly, the five-block-size configuration (1 kilobyte, 8 kilobytes, 64 kilobytes, 1 megabyte, 16 megabytes), which is the rightmost group on each graph, is chosen. Clustering had little effect on the large file environments and improved performance in the time-sharing environment, so the clustered configuration was selected. In four of the six cases, the grow factor of one provided better

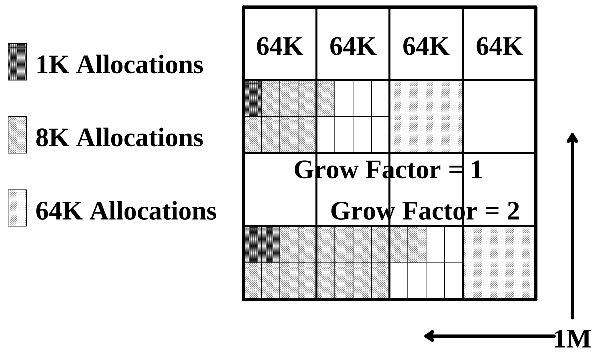


Figure 3-4: Interaction of Contiguous Allocation and Grow Factors. Because the total file length is not a multiple of the new block size, a seek is required when the block size grows.

throughput than the grow factor of two, so the policy with the grow factor of one is selected, with the understanding that it will penalize sequential performance for the time-sharing workload. This configuration is represented by the leftmost bar in the rightmost group of each graph.

3.3.3. Extent-Based Systems

In the extent-based models, every file has a single extent size associated with it. Each time a file grows beyond its current allocation, additional disk storage is allocated in units of this extent size. As in the restricted buddy policy, the disk system is viewed as a linear address space. However, in this model, an extent may begin at any disk offset. When an extent is freed, it is coalesced with any adjoining free extents.

Workload	Number of Ranges	Range Means
TS	1	4K
	2	1K, 8K
	3	1K, 8K, 1M
	4	1K, 4K, 8K, 1M
	5	1K, 4K, 8K, 16K, 1M
TP/SC	1	512K
	2	512K, 16M
	3	512K, 1M, 16M
	4	512K, 1M, 10M, 16M
	5	10K, 512K, 1M, 10, 16M

Table 3-6: Extent Ranges for Extent-Based File System Simulation..

The parameters which define a file system in the extent-based model are the allocation policy and the variation in the sizes of the extents. The allocation policy indicates how to select the next extent for allocation. Both a first-fit and best-fit algorithm are simulated.

In order to simulate the variation in the size of extents, *extent ranges* are used. In extent-based systems, such as MVS [IBM] , users specify extent sizes when they create files. In the simulations, when a file is created, its extent size is chosen from a distribution called an extent range. An extent size range is a normal distribution with a standard deviation of 10% of the mean. For example, an extent range around 1 megabyte with 1 kilobyte disk units would produce a normal distribution of extent sizes with mean 1 megabyte and standard deviation of 102 kilobytes. To assess the impact of the variation in extent-sizes, the simulation is run with varying numbers of the extent ranges. Table 3-6 shows the extent ranges simulated.

As the number of extent ranges increases, one expects to see increased fragmentation since a more diverse set of extent sizes are being allocated, but the results do not support this. Instead, across all extent ranges, both internal and external fragmentation is below 4%, independent of the number of extent ranges. One likely explanation is that the ratio of large files to small files is constant in these simulations. As a result, once large extents are allocated they do not become fragmented later, because requests for small extents may be satisfied by already fragmented blocks. This also explains why best fit consistently result in less fragmentation.

One might expect throughput to be insensitive to the selection of best fit or first fit since, in both cases, files are read in the same size unit. Figure 3-5 shows the application and sequential performance results for the extent-based polices and confirms this intuition. In general, first fit demonstrates better performance due to the clustering that results from the tendency to allocate blocks toward the ‘‘beginning’’ of the disk system.

The key to the small changes in performance is the average number of extents per file for the different workloads and extent ranges. These numbers are summarized in Table 3-7. Since the workload with the minimum average number of extents requires the fewest seeks, one would expect to see the best performance for that workload. The super-computer and transaction processing workloads behave as expected (the first two graphs in the right-hand side of Figure 3-5), but the time-sharing workload does not. Further inspection indicates that the ratio of small to large files alters this result. Since most of the files in the time-sharing environment are small, they can be allocated in one or two 4 kilobyte extents. The larger files require 24 extents (96 kilobyte files with 4 kilobyte extents). However, the larger files consume more disk space and take longer to read and write. As a result, the time spent processing large files is greater than the time spent processing small files. Therefore, in the configurations where the large files have fewer extents (12 extents in the systems that use 8-kilobyte extents for these files), the overall throughput is higher.

In selecting the configuration to compare in Section 3.5, first fit allocation is chosen since it consistently provides better performance than best fit. For the transaction processing and super-computer workloads simulated, the three range size configuration results in the highest sequential performance. Although this configuration does not offer the best performance for the timesharing workload, it is within 10% of the best performance. This configuration is represented by the right-hand bar in the middle group of each graph.

3.3.4. Fixed-Block Allocation

The last of the allocation policies is a simple fixed-block algorithm used as a control to establish how much of an improvement may be derived from the multiple-block systems described. When small files are the predominant part of the workload (as in the time-sharing workload), a small block size of 4 kilobytes is used. Where an abundance of large files are present as in the super-computing and transaction processing workloads, a larger, 16 kilobyte block size is used.

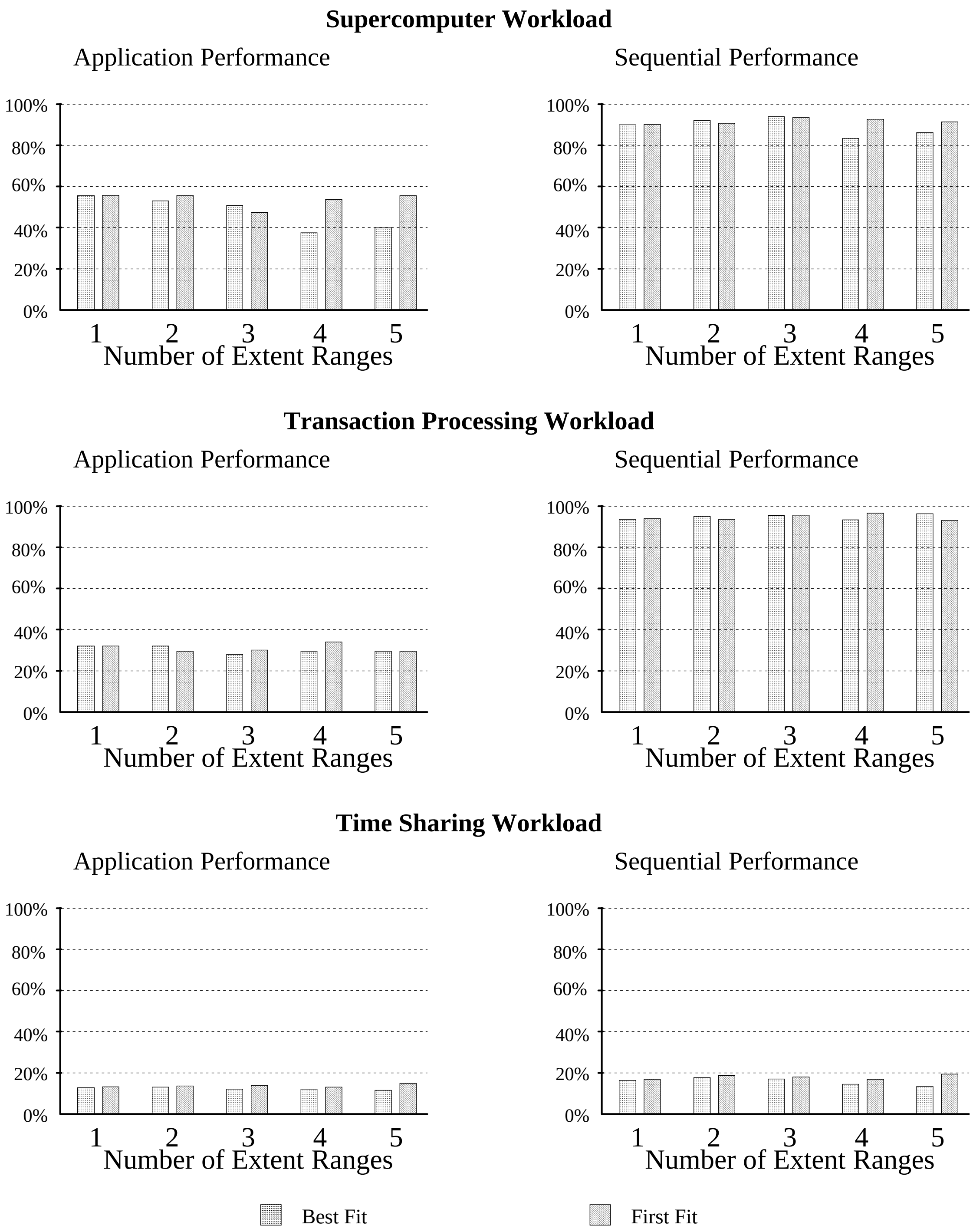


Figure 3-5: Application and Sequential Performance for the Extent-based System.

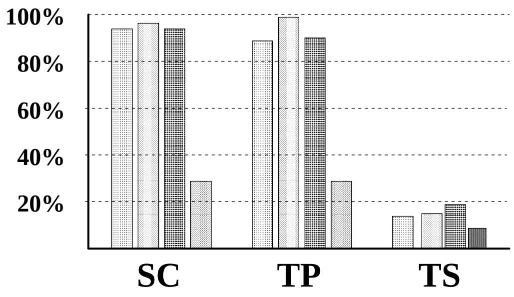
Average Number of Extents Per File			
Number of Extent Ranges	SC	TP	TS
1	162	267	5
2	124	13	9
3	97	12	9
4	151	14	7
5	162	108	6

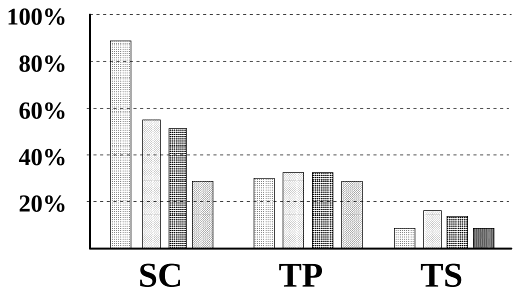
Table 3-7: Average Number of Extents per File.

3.4. Comparison of Allocation Policies

As we've seen in the preceding sections, all the allocation policies except for the buddy system yield satisfactory fragmentation. As a result, this section focuses on the application and sequential performance.

Figure 3-6 shows the sequential performance of the four allocation policies discussed in Section 3.3. As expected, all of the multiblock policies perform better than the fixed-block policy due to the ability to read and write large, contiguous blocks. On the large file applications (SC and TP) all the large-block policies achieve close to the maximum throughput. In the time-sharing environment, none of the policies succeed in pushing the system above 20% utilization due to the presence of many small files. However, the extent-based policy can respond to this burden most effectively since each file is limited to a small number of extents.





Buddy Allocation | Fixed Block (16K)
Restricted Buddy | Fixed Block (4K)
Extent Based

Figure 3-6: Sequential Performance of the Different Allocation Policies.

Figure 3-7: Application Performance of the Different Allocation Policies..

In the application performance (Figure 3-7), the results are similar. However, there are two points to note. First, in the super-computer environment, the buddy system performs substantially better since, for large files (over 100 megabytes), it is using substantially larger block sizes (64 megabytes). In the transaction processing environment, all the policies are limited by the random reads and writes to the large data files.

3.5. Conclusions

File systems with variable block sizes can substantially improve performance by allowing transfers to and from the disk in large, contiguous units. In the large file environments such as super-computer applications, these large-block policies provide up to 250% better throughout than a simple fixed-block policy. Even for workloads like the transaction processing environment, which are dominated by small reads and writes to large files, there is a small (10%) improvement. While the large blocks do not benefit the small file environment greatly, they do not hinder it either in terms of performance or fragmentation. Therefore systems with both extremely large and extremely small files are likely to be able to derive this improved performance without handicapping the efficiency of small files' retrieval.

This result suggests that time-sharing environments could benefit significantly from these allocation techniques. Such systems could then effectively compete with systems designed with database or super-computer applications in mind, without hindering the small file performance. Empirical evidence in [ROSE91] and [ROSE92] shows that log-structured file systems also provide these benefits in a time-sharing environment. Since LFS can guarantee sequential layout for large files read and written in their entirety, one might expect that it will also perform well on the super-computer workload. However, it is not clear how well LFS can support the transaction processing workloads. Chapter 4 will explore the comparison of LFS and read-optimized file systems in the case of transaction processing workloads.

Chapter 4

Transaction Performance and File System Disk Allocation

This chapter considers the use of a write-optimized file system, specifically a log-structured file system, in a transaction processing environment. The goals of this chapter are twofold: to understand the tradeoffs between using a conventional read-optimized file system and a write-optimized file system for transaction processing, and to characterize the performance of embedding a transaction manager in the operating system. As discussed in Chapter 2, [KUM87] and [KUM89] show that embedded support is inferior to traditional, user-level support, but [SELT90] shows that embedded support can be as good as user-level support. This chapter will explain this result by analyzing five models of transaction processing, isolating the critical resources, and stressing each model in each dimension, enabling a characterization of the performance of each model across a wide range of configurations.

The rest of this chapter is organized as follows. First, the write-optimized (log-structured) file system is described. Then the overview of the simulation is presented. Next, the simulation model and the different models of transaction management are discussed. Finally, the simulation results are presented.

4.1. A Log-Structured File System

A log-structured file system is a hybrid between a simple, sequential database log, and the traditional UNIX file system. Like a database log, it performs all writes sequentially. Like a UNIX file system, it has index structures to support efficient random retrieval. The index structure contains the disk addresses of some number of *direct*, *indirect*, and *doubly indirect* blocks. Direct blocks contain data, while indirect blocks contain the disk addresses of direct blocks, and doubly indirect blocks contain disk addresses of indirect blocks. For the purposes of this chapter, the index structures and both single and double indirect blocks are referred to as *meta-data*.

While conventional UNIX file systems allocate disk space to optimize for sequential access to a single file, an LFS allocates disk space dynamically, optimizing write performance. In a conventional file system, the blocks within a file are assigned disk addresses, and each time a block is modified, the same disk block is overwritten. As a result, writes to different files often cause a seek, and writes to different blocks of a file may also cause a seek. In an LFS, a large number of modified data pages, the meta-data describing them, and a segment summary block are written sequentially in a single unit, called a segment [ROSE90]. Note that while the index structures in a UNIX file system occupy fixed places on disk, the index structures are appended to the log with their data in an LFS. In order to locate these index structures later, the address of each file's index structure is recorded in a data structure called the *inode map*. Figure 4-1 shows the allocation of three files in a log-structured file system.

When a file is written, the new data blocks are appended to the log, and the index structure and indirect blocks are modified (in memory) to contain the new disk address of the newly

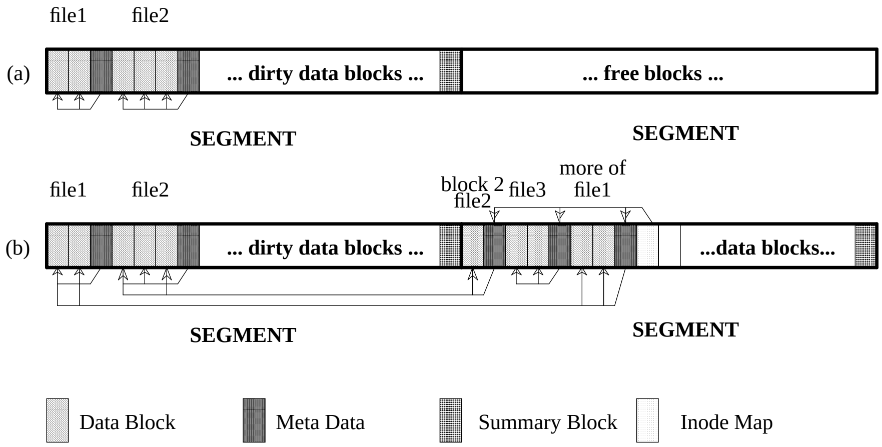


Figure 4-1: A Log-Structured File System. In figure (a), two files have been written, file1 and file2. The meta-data block following each file contains that file's index structure. In figure (b), the middle block of file2 has been modified. A new version of it is added to the log, as well as a new version of its meta-data. Then file3 is created, causing its blocks and meta-data to be appended to the log. Next, file1 has two more blocks appended to it. These two blocks and a new version of file1's meta-data are appended to the log. Finally, the *inode map*, which contains pointers to the meta-data blocks, is written.

written block. Periodically, the file system writes all the dirty meta-data (index structures and indirect blocks) to disk and updates the inode map to reflect the new location of modified index structures. This *checkpointing* provides a stable starting point from which the file system can be recovered in case of system failure. The location of the latest checkpoint is redundantly written in fixed places on the disk to facilitate recovery.

Recovering a log-structured file system is similar to standard database recovery [HAER83]. It consists of two parts: initializing all the file system structures from the most recent checkpoint and then *rolling forward* to incorporate any modifications that occurred after the last checkpoint. The roll forward phase consists of reading each subsequent segment summary block and updating the file system state to reflect the contents of the segment. Each segment summary block includes a pointer to the next segment written, a timestamp, and a file identification number and logical block number for each block in the segment. The forward pointers facilitate reading from the last checkpoint to the end of the log and the timestamps are used to identify the segments written after the checkpoint. The file identification numbers are used to index into the inode map, and the logical block numbers are used to update the file's meta-data so that the file index structure includes the blocks in the segment. As is the case for database recovery, the recovery time is directly proportional to the interval between file system checkpoints.

Since the structure described is an append-only log, the disk system will eventually become full, requiring a mechanism to reclaim space. If there are files which have been deleted or modified, some blocks in the log will be ''dead'' (those that belong to deleted files or that have been superceded by later versions). A cleaning process reclaims segments from the log by reading a segment, discarding ''dead'' blocks, and appending any ''live'' blocks to the log. In this manner, space is continually reclaimed [ROSE91].

There are two characteristics of a log-structured file system that make it desirable for transaction processing. First, a large number of dirty pages are written contiguously. Since only a single seek is performed to write out these dirty blocks, the ‘‘per write’’ overhead is much closer to that of a sequential disk access than to that of a random disk access. Taking advantage of this for transaction processing is somewhat similar to the database cache discussed briefly in Section 2.1.2.2 and in more detail in [ELKH84]. While the database cache technique writes pages sequentially to a cache, typically on disk, blocks in the log still need to get written back to the ‘‘real’’ database. In an LFS environment, these blocks are still written sequentially as they are in a log, but they also become data blocks in the ‘‘real’’ database.

The second characteristic of a log-structured file system that makes it desirable for transaction processing is that the file system is updated in a ‘‘no-overwrite’’ fashion. Since data is not overwritten as part of the update process, before-images of updated pages exist in the file system until they are reclaimed by the cleaner. This feature makes it possible to avoid writing separate log records during transaction processing.

4.2. Simulation Overview

The goal of this study is to answer three questions. First, how does the performance of a log-structured file system compare to that of a conventional file system on a transaction processing workload? Second, how does the performance of operating system transaction management compare to that of user-level transaction management? Third, does the answer to the last question change depending on the file system? In order to answer these questions, four systems are simulated -- one with user-level transaction processing and a read-optimized file system (USER-RO), one with user-level transaction processing and a write-optimized, log-structured file system (USER-LFS), one with operating system transaction processing and a read-optimized file system (OS-RO), and one with operating system transaction processing and a write-optimized file system (OS-LFS). A fifth model (LFS-NOLOG) that exploits the logging nature of log-structured file systems is also simulated.

In order to understand the salient features of each model, they are analyzed in CPU-bound, disk-bound, and lock-bound environments. By focusing on one component of the performance (CPU, disk, or locking) in each simulation, we can identify the critical performance issues. Figure 4-2 depicts the three dimensions of the simulation study.

As discussed in Chapter 2, user-level transaction systems typically offer better performance than operating system transaction systems, but operating system transactions are available to a wider class of applications. The difference in performance is due to three factors: the system call overhead, the operating system’s inability to perform special-purpose locking for structures such as B-Trees, and the operating system’s need to log data at a physical level. The system call overhead is a factor because processes must make system calls to request transaction service in an embedded model while in user-level models, they can communicate via shared memory. The lack of special-purpose locking is a disadvantage in high contention environments, and the physical log requires more disk space. If operating system transaction management could overcome some of these performance barriers and still offer a more flexible alternative than user-level systems, it would be an attractive alternative.

4.3. The Simulation Model

The simulations use a workload characterized by short-running transactions and are driven by a stochastically generated workload. The database consists of a single data file with a variable number of B-Tree indices. Its size and *fillfactor* (the fraction of each page containing valid data) are simulation parameters.

The workload consists of M concurrent processes issuing a potentially infinite stream of parameterized transactions. At initialization, M transactions are created, and each time a

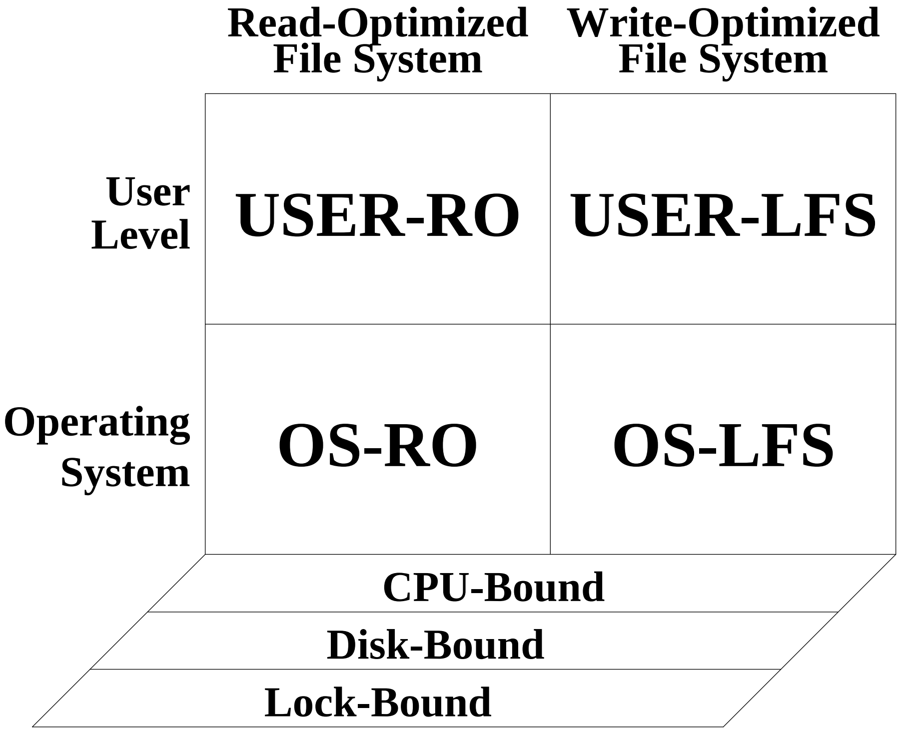


Figure 4-2: Simulation Overview. The simulation study compares read-optimized and write-optimized file systems with both user-level and operating system transaction management. Each of the four resulting models are analyzed under CPU-bound, disk-bound, and lock-bound conditions.

transaction commits or aborts, a new transaction is created. A transaction is defined to be a sequence of retrieve, update, insert, and delete operations. Each retrieve and update operation affects a single data page and a search path through a single index. A search path consists of an access to one page in each level of the B-Tree, culminating with a leaf page. An insert or delete operation affects a single data page and a search path through each index, since it is presumed that a key must be inserted/deleted into/out of each index.

A number of operations, O, uniformly distributed over $[l - 0.25l, l + 0.25l]$, where l is the average transaction length, is generated. A second parameter, U, determines what percent of the O operations modify the database, the rest being read-only operations. Finally, a third parameter f identifies what percentage of the modify operations are inserts or deletes (as opposed to updates). Inserts and deletes can be treated identically as they both require modifying the data and all indices. A transaction may then be defined as:

O total operations composed of:

(1-U)O	retrieves
fUO	inserts/deletes
(1-f)UO	updates

Each operation of a transaction is processed in the following manner. A data page is selected from a distribution described by two parameters d and a. The parameter d indicates what percent of the database gets a percent of the accesses. For example, $d=20$ and $a=80$ means that 80% of the accesses are distributed uniformly across 20% of the database. Once a data page is selected, it is locked, read from the disk or the buffer pool, and left locked until transaction commit time. To simulate index traversal, using the same distribution as was used for the data file, one page is selected from each level of the B-Tree. These pages are locked, read, and unlocked either at completion of the operation (for data manager models) or at transaction commit time (for embedded models). As soon as one operation completes, the next operation begins. When all the operations have completed, a synchronous write forces the log to disk.

The total database size is derived from the *dbsize*, *pagesize*, and *fillfactor* parameters. *Dbsize* defines the size (in megabytes) of the data file. The number of records in the database is determined by using the *fillfactor* parameter, which defines how much data is on each page. Then, using the number of records, the *fillfactor*, the *pagesize*, and the size of a key (16 bytes), the number of index pages is determined, using the formulas below.

$$L=\frac{R * K}{F * P} \quad \text{and} \quad L_i=\frac{L_{i+1} * K}{F * P}$$

where:

L is the number of leaf pages
L_i is the number of B-Tree pages at level i
R is the number of records in the data file
K is the key size
F is the fillfactor
P is the pagesize

Once the size of each index has been calculated by summing L_i, the number of indices is multiplied and data file size is added to yield the total database size.

The buffer pool size is defined to be 10% of this total database size. The buffer pool uses an LRU replacement algorithm and flushes dirty blocks to disk asynchronously. In both [KUM87] and [KUM89], the buffer pool is sized in terms of a number of pages. This penalizes simulations with a smaller page size by providing them less main memory. Keeping the amount of main memory constant and reducing the page size can improve performance by more than 25% in high contention environments.

Table 4-1 summarizes the number of instructions required to perform each operation. The instruction count for locking includes both the lock and unlock actions. In the embedded models, it is assumed that a system call is required to obtain a lock, so the actual cost of a lock is a function of the number of instructions for both a system call and a lock. It is assumed that all unlocking may be performed by a single system call at transaction commit time. Therefore, the CPU-boundedness of a configuration may be adjusted by setting only the *available CPU* parameter.

operation	number of instructions
lock	1000
syscall	500
retrieve	7000
update	12000
insert/delete	18000

Table 4-1: CPU Per-Operation Costs.

The last set of parameters controls deadlock detection and recovery. The deadlock detector runs every *deadlock* seconds, aborting transactions which have been waiting longer than the timeout interval. In order to limit recovery time and allow log reclamation, checkpoints are taken every *chkpt* seconds. At checkpoint time, all dirty pages are forced to disk and creation of new transactions is inhibited until all active transactions have committed. Table 4-2 summarizes the simulation parameters and their default values.

4.4. Transaction Processing Models

This analysis considers five models of transaction processing. The first is a conventional data manager on a traditional, read-optimized file system. The second is the same data manager on a write-optimized file system. The third embeds transaction support in the read-optimized file system. The fourth and fifth both embed transactions in a write-optimized file system. The fourth uses traditional write-ahead logging in the operating system while the fifth takes advantage of the log-structured file system's ‘‘no-overwrite’’ policy to obviate the need for a separate log.

Statistical Parameters		
parameter	description	default
runlen	Transactions per run	10000
nruns	Runs per data point	5
Workload Characteristics		
parameter	description	default
O	Avg ops per transaction	16
U	% update operations	.25
f	% insert/delete	.50
d/a	Request distribution	50/50
I	Number of indices	5
dbsize	Mbytes in the data file	1024 (1G)
bufsize	Buffer pool size	10% of db
fillfactor	Valid fraction of page	.70
recsize	Length of data records	100 bytes
System Parameters		
parameter	description	default
cpu_speed	Processor speed (in MIPS)	10
disks	Number of disks	10
users	Degree multiprogramming	20
pagesize	Page size (in bytes)	4096
spagesize	Subpage size	128 bytes
deadlock	Deadlock detector interval	5 sec
chkpt	Checkpoint interval	5 min

Table 4-2: Simulation Parameters.

4.4.1. The Data Manager Model

In the data manager model, detailed knowledge of the structure of the database is assumed. For example, logging is performed at a logical, rather than a physical level, allowing log records to contain only the modified data instead of the whole containing page. Using special concurrency control protocols, facilitating high degrees of parallelism [BAYER77], the data manager only needs to hold index locks during the physical manipulation of the index page (on the order of a few thousand instructions), providing superior performance in environments with high lock contention.

The sequence of events for accessing a random record in the database is as follows. First, a keyed lookup is performed. This requires traversing the non-leaf pages of a B-Tree by obtaining a read-lock on each page, finding the next page to access, and releasing the read-lock. When a leaf page is reached, the data page is locked and accessed. In the case of an update (updates change the record and one index, while creates and deletes update all the indices) a write-lock is obtained on the leaf page of the B-Tree. Then, the update is logged, by recording both a before- and after-image of the record, the index page and data page are modified, and the index locks are released.

A transaction can be decomposed into operations whose cost may be expressed as a combination of logging, I/O, and locking costs. In the data manager models, group commit can be used to accumulate enough data to fill a track so that the logging cost is proportional to the record size. Since the access pattern is random, in a read-optimized file system the I/O cost for both reading and writing is proportional to the random access time of the disks (approximately 28.3 milliseconds)[3]. On a write-optimized file system, the reads are also performed randomly, but the writes are all performed sequentially (Section 4.1 explained how this is achieved). Finally, since the data manager has its own lock manager, the locking cost is strictly a function of the number of locks and is independent of any system call overhead.

4.4.2. The Operating System Model

As the operating system knows nothing about the internal structure of files, it cannot distinguish between data and index updates. In order to guarantee serializability it must perform strict two-phase locking [GRAY76] on physical entities (pages). If there are few conflicts in the index, then page locking will be the least expensive locking granularity. However, as contention increases, page locking in the index will limit performance, so the simulation model allows for subpage locking as well.

Assume that there are S subpages per page. To traverse a B-Tree, $\log_2 S$ subpages, selected uniformly from the filled subpages within the page, are locked. This models searching for a key within the page.[4] To modify a leaf page, one of the selected subpages is write-locked. If a key is being deleted, the data that follows is copied to reclaim the space. If the key is being inserted, the data following it is shifted to make room for the new key. Figure 4-3 shows these operations. As a result, all the subpages after the selected subpage must also be write-locked. This requires the operating system to obtain multiple write-locks (on average half the number of filled subpages per page) for each B-Tree page modified as compared to the data manager's one lock.

Since all the transaction support is provided in the operating system, each lock request requires a system call. In simulating the embedded models, the time required to perform a system call is added to each lock request while in the data manager models, no system call overhead is added. This puts an artificially high penalty on the embedded models since, in practice, the data manager will incur system call overhead each time a page that is not resident in the buffer pool is requested.

---

[3] All disk times are based on the performance specification of the Fujitsu Eagle M2361A [FUJI84].

[4] The log_2 assumes a binary search is used to locate the correct subpage.

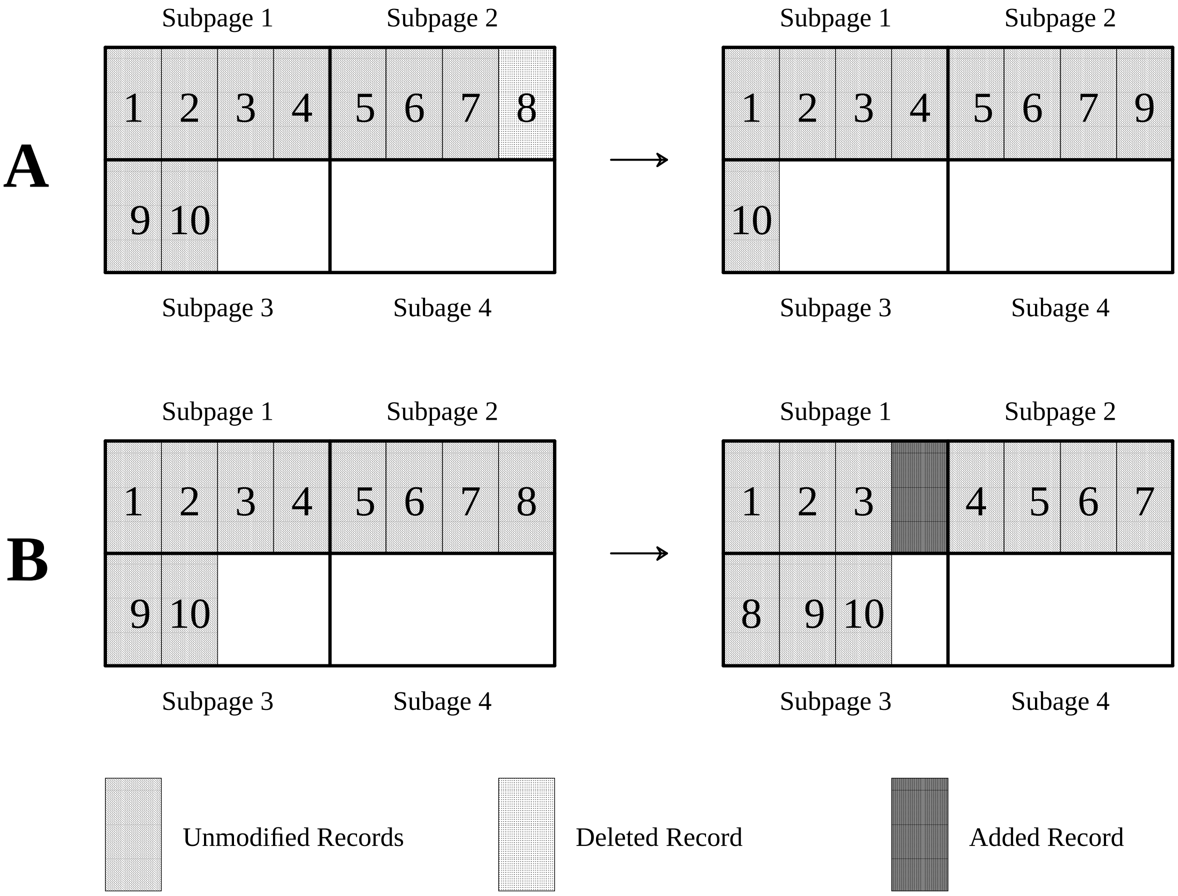


Figure 4-3: Additions and Deletions in B-Trees. In Figure A, record 8 is being deleted. Records 9 and 10 are copied to reclaim space on the page. Both pages 2 and 3 must be locked. In Figure B, a new record is being inserted between records 3 and 4. Records 4-10 are copied to make room for the new record, requiring locks on subpages 1, 2, and 3.

The final difference between the data manager and the operating system embedded model is the amount of logging information. Since the operating system cannot perform logical logging, it must resort to physical logging and save both before- and after-images of each subpage that is modified. In the case of inserts and deletes, this number may become quite large since multiple subpages per index page are modified.

As before, the transaction cost is decomposed into logging, I/O, and locking costs. This time, the logging cost is proportional to the size of a subpage, the I/O costs are proportional to the random disk access time, and the locking cost is a function of the number of locks, the number of subpages per page, and the system call overhead.

4.4.3. The Log-Structured File System Models

There are three log-structured file system models. The first is a user-level system, identical to the user-level data manager model, except the underlying file system is an LFS. The second is an embedded model identical to the operating system model, except that its underlying file system is an LFS. Both of these have locking and logging costs identical to the data manager and operating system models respectively, but the I/O component of the transaction cost is proportional to the random disk access time for reading and the sequential disk access time for writing. Neither model includes any cost for the cleaner.

The third model takes advantage of both the sequential nature of writes and the ‘‘no-overwrite’’ policy of the log-structured file system. Instead of logging before- and after-images of the subpages being modified, all dirty pages are forced to disk at commit time. Since a single page is composed of multiple subpages, the page may contain subpages modified by more than one transaction. When one of those transactions commits, the page is written, and subpages for uncommitted transactions may also be written to disk. A small log which records the location of the previous and current versions of all dirty, uncommitted subpages is necessary to guarantee that these uncommitted transactions can be aborted. This logging information must be forced to disk before the dirty pages themselves. The difference between these last two embedded models is that the latter has very small log records (16 bytes) and the logging overhead is proportional only to the number of subpages modified rather than to both the number and size of the subpages. This model also ignores cleaner overhead.

4.4.4. Model Summary

In the discussion that follows, USER-RO refers to the data manager on a read-optimized file system, and USER-LFS refers to the data manager on a log-structured file system. OS-RO refers to transaction support embedded in a read-optimized file system, OS-LFS refers to embedded support in a log-structured file system using a full log, and LFS-NOLOG refers to embedded support in a log-structured file system, using the file system in place of a traditional log. For each component of transaction cost (logging, I/O, and locking), Table 4-3 indicates the parameters

Description	Label	Logging function of	I/O (read)	I/O (write)	Locking function of
User-Level Transaction Manager Read-Optimized File System	USER-RO	# updates record size	random	random	# locks
User-Level Transaction Manager Write-Optimized File System	USER-LFS	# updates record size	random	seq	# locks syscall cost
Embedded Transaction Manager Read-Optimized File System	OS-RO	# updates subpage size	random	random	# locks syscall cost subpages/page
Embedded Transaction Manager Write-Optimized File System	OS-LFS	# updates subpage size	random	seq	# locks syscall cost subpages/page
Embedded Transaction Manager Write-Optimized File System	LFS-NOLOG	# updates	random	seq	# locks syscall cost subpages/page

Table 4-3: Comparison of Five Transaction Models. The transaction cost is decomposed into its logging, I/O, and locking components and each column indicates upon which parameters this cost depends. For example, the logging cost is dependent upon the number of updates in all the models, but upon the record size only in the user-level models and the subpage size only in OS-RO and OS-LFS models.

upon which the component is dependent for each model.

4.5. Simulation Results

The three potential performance bottlenecks are the CPU, the disk system, and lock contention. By isolating each of these resources, all five systems can be stressed in each dimension, resulting in a characterization of the performance of each model across a wide range of configurations. For all the simulations reported, the configuration consists of a one gigabyte data file with 1.2 gigabytes of index information (five indices). This data is striped across ten small, inexpensive disks to achieve parallelism in the I/O subsystem. For both data and indices, block *i* is assumed to reside on disk i % 10. Varying the CPU speed and the locality of accesses produces CPU-bound, disk-bound, and lock-bound configurations.

To verify the simulation results, the upper and lower limits for each configuration were computed analytically. Then, the simulation results were plotted, checking that the limits approached those from the analytic model. Each of the data points represents five runs of 10000 transactions each. The variance across the five runs is approximately 1% of the average and yields 95% confidence intervals of approximately 2%.

4.5.1. CPU Boundedness

To create a CPU-bound environment, the available CPU is set low, to 1 MIPS. The available CPU means the amount of processing power dedicated to the operations being analyzed (obtaining/releasing locks, issuing a system call, searching a page, and modifying data), ignoring overhead for query processing, communication, setting up I/O, scheduling, etc. These other overheads are ignored, so that the simulations can isolate those aspects of the system that differ (locking cost, logging cost, CPU utilization), focusing on how each impacts performance.

In order to guarantee that the configuration is CPU limited and not contention limited, the access pattern is uniform. This yields a probability of conflict of approximately 5%, so there is no need to set the locking granularity (or subpage size) any smaller than the page size for the embedded models. This differs from the simulations in [KUM87] that model the 801 hardware locking [CHAN88], which always performs subpage locking on 128 byte subpages.

Figure 4-4 shows the results of varying the degree of multiprogramming until the CPU becomes saturated. In this configuration the two data manager models provide better performance than any of the embedded systems. Whereas Kumar found this difference to be 30% or more in a CPU-bound configuration, these results show that at saturation the difference in throughput between the user-level models and the embedded models is approximately 17% ($\frac{T_{USER-RO} - T_{OS-RO}}{T_{USER-RO}}$), and the difference between USER-LFS and either OS-LFS or LFS-NOLOG is 20% ($\frac{T_{USER-LFS} - T_{OS-LFS}}{T_{USER-LFS}}$). These results are different from Kumar's because our model requires only one lock per B-Tree level while Kumar's required four. Therefore, the difference in performance between the user-level and embedded models is due only to the number and cost of the system calls required by the embedded models.

Using the cost components detailed in Table 4-3, the CPU costs for the data manager and embedded models can be expressed as: $T_{os} = N(L + S) + C$ and $T_{user} = LN + C$ and throughput is proportional to $\frac{1}{T}$, so the relative performance, in terms of throughput, may be expressed as:

$$T_{os} = \left(\frac{LN + C}{N(L+S) + C} \right) T_{user} \quad OR \quad T_{user} = \left(1 + \frac{NS}{LN+C} \right) T_{os}$$

where

N is the number of locks required,

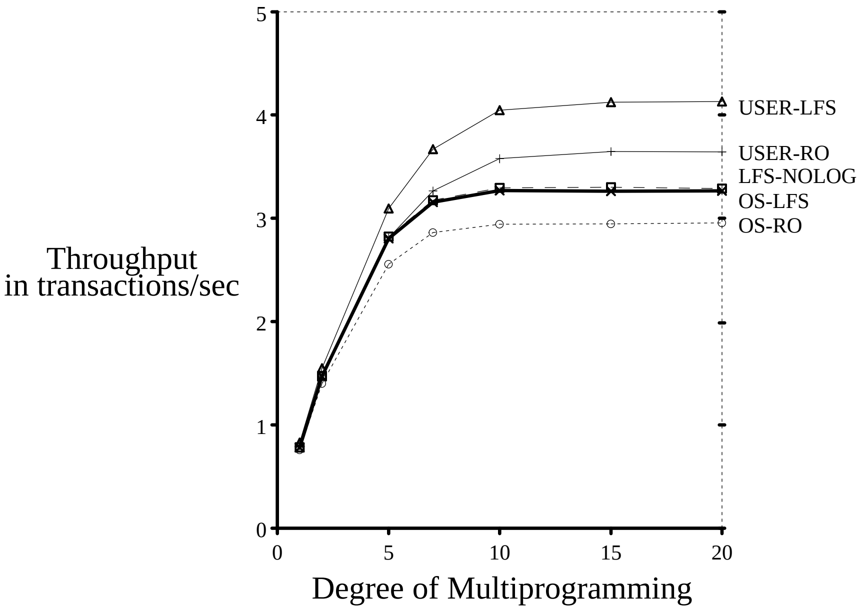


Figure 4-4: CPU Bounding Under Low Contention. The degree of multiprogramming is varied in a low contention configuration. Since the embedded models incur a system call per lock, the user-level models (USER-RO and USER-LFS) outperform the embedded models (OS-RO, OS-LFS, LFS-NOLOG). What is unexpected is that although the configuration is CPU-bound, the file system still impacts the resulting performance as is evidenced by the difference between the RO lines and the LFS lines.

L is the CPU overhead for acquiring a lock,
C is the CPU overhead for searching and modifying data pages.
S is the cost of a system call (in milliseconds).

For the workload simulated, T_{user} is $1.2T_{os}$ or $(1+.4S)T_{os}$. It is apparent that the cost of a system call has a tremendous impact on the performance. Figure 4-5 graphically depicts this difference in performance as the cost of a system call is varied. In the preceding simulation a .5 millisecond overhead for system calls was used, yielding a 17-20% difference in performance between the user-level and embedded models. At .25 millisecond (250 instructions), the difference between the data manager and embedded models drops to 12%.

Surprisingly, although this configuration is nearly CPU-bound, the log-structured file system models provide better performance than the read-optimized file system models. Comparing the USER-LFS performance with the USER-RO performance, there is a gap of nearly 12% (4.1 tps v.s. 3.6 tps), and comparing LFS-NOLOG with OS-RO, there is a gap of 10% (3.3 tps vs. 3.0 tps). In each of these situations, the CPU cost for both configurations is the same. Since the configuration is not disk-bound, the better performance of the log-structured file system is unexpected. Upon closer inspection, the read-optimized file systems are achieving only 87% CPU utilization but 50% disk utilization while the write-optimized models are achieving 99% CPU utilization and 33% disk utilization.

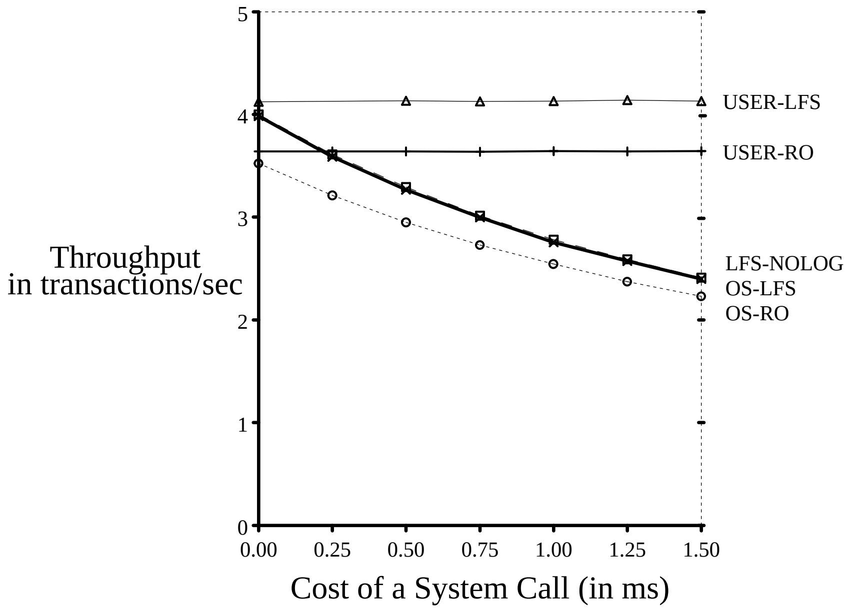


Figure 4-5: Effect of the Cost of System Calls. As system calls become more costly (in terms of CPU cycles), the difference in performance between the data manager and the embedded models widens.

Examination of what happens as dirty blocks are flushed to disk explains how the disk utilization affects throughput. In the read-optimized models, flushing a dirty block busies the disk for the time of a random access. When dirty data blocks are written, an incoming read request may be delayed for up to 28.3 milliseconds. Even if these writes are attempted during idle disk cycles, subsequent read requests may still queue up behind the writes and be delayed. On the other hand, when the log-structured file system models flush dirty blocks, they write a large number of blocks at sequential speed (1.99 milliseconds per 4-kilobyte block). Therefore, the potential delay incurred per block flushed is much less. Additionally, each time that the log is forced to disk (or the dirty file blocks in the embedded model), any other dirty buffers that happen to be in the cache are also written for little cost. As a result, the LFS-based models rarely wait to evict pages from the buffer pool. Looking at this another way, in the read-optimized models, the CPU utilization peaks at approximately 87%, because queueing at the disks causes a convoy affect, preventing the CPU from being used efficiently. In contrast, the CPU utilization for the log-structured file system models approaches 100%. So, even when the disks are not the critical resource, the difference in write performance of the disk systems impacts the resulting throughput.

4.5.2. Disk Boundedness

By increasing the available processing power, the configuration becomes disk-bound. Once again, the degree of multiprogramming is varied to determine a saturation point. These results are shown in Figure 4-6. As expected, the log-structured models provide the best performance, by approximately 23% (9.4 tps for USER-LFS and 7.2 tps for USER-RO). Furthermore, although the configuration is disk-bound, the size of the log does not have a significant impact on performance. Both the user-level and embedded models exhibit nearly identical performance, even though the operating system maintains a much larger log. As in the CPU-bound case, these results differ dramatically from [KUM87]. He found that in disk-bound configurations the data manager out-performed the operating system embedded model and attributed this difference to the size of the log. Although the OS systems keep a much larger log than the USER systems, their performance is nearly identical as shown by the overlapping lines in Figure 4-6. Similarly, USER-LFS, OS-LFS and LFS-NOLOG exhibit nearly identical performance although these models have different sized logs as well. To understand this phenomenon, consider how the total transaction time is apportioned to different operations. Logging occurs at sequential speed and makes up only a small fraction (less than 1.2%) of the total I/O time, so the total transaction time is dominated by the random read time (more than 73% of total I/O time). Since Kumar ignored the time required to randomly write dirty blocks back from the cache, his overall I/O time was much smaller, thereby making the log write time much more important.

Having analyzed the extremes of disk-boundedness and CPU-boundedness, the region in between is analyzed. Varying the available CPU yields the results shown in Figure 4-7. Between any two models, there are two factors that contribute to the performance differential: the file system and the location of transaction support (user-level or operating system). At 1 MIPS, the CPU-bound configuration, the file system component accounts for a 10-12% difference in performance (the difference between USER-LFS and USER-RO or OS-LFS and OS-RO) and the location of transaction support accounts for a 19-20% difference (the difference between OS-RO and

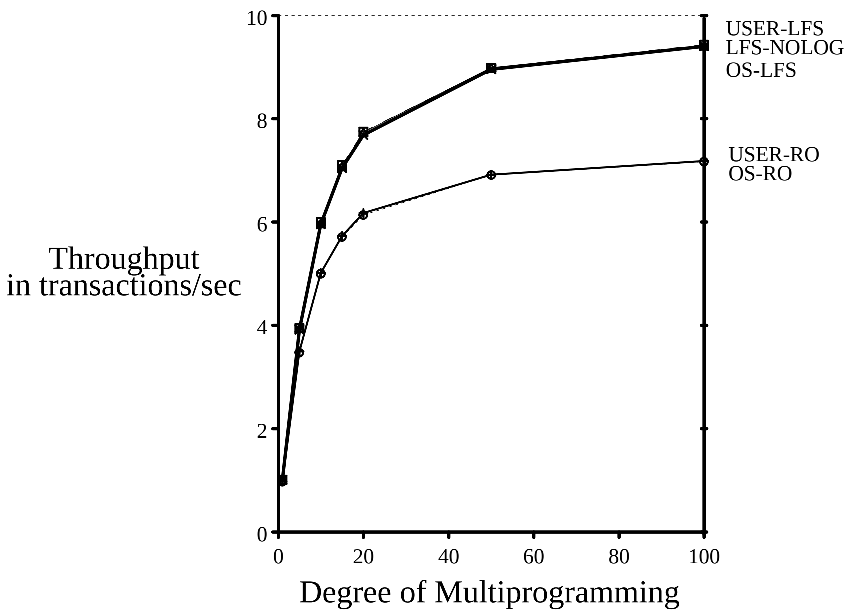


Figure 4-6: Disk Bounding Under Low Contention. Since there is sufficient CPU power to support the more expensive embedded systems, the file system determines performance, and the write-optimized file system provides superior performance to the read-optimized one. Surprisingly, the number of bytes logged does not affect the performance as the USER-LFS, OS-LFS, and LFS-NOLOG all exhibit the same performance.

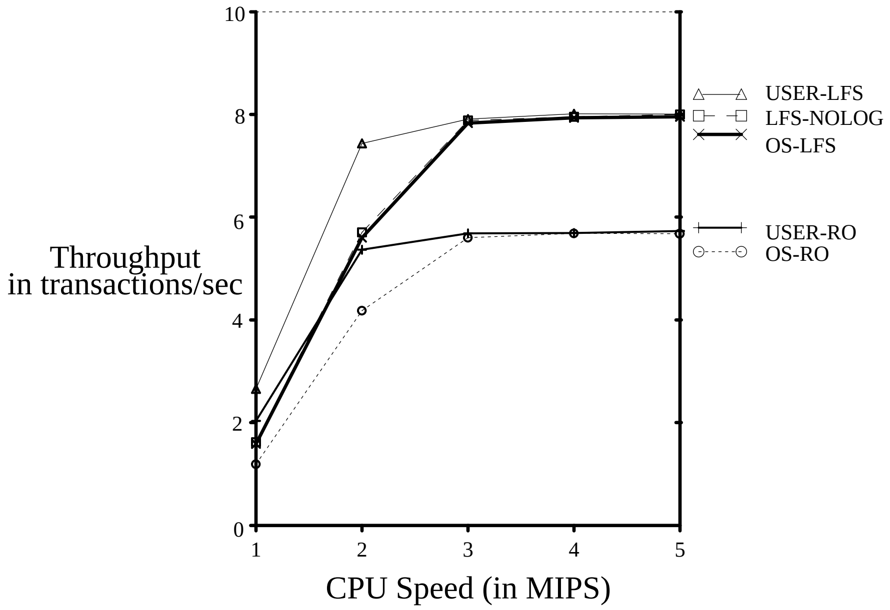


Figure 4-7: Effect of CPU Speed on Transaction Throughput. Increasing CPU speed moves the configuration from a state of CPU-boundedness to disk-boundedness. Even before the systems become completely disk-bound (at 3 MIPS), the major factor contributing to the performance differential is the file system as opposed to whether transaction support if provided in the operating system or at user-level.

USER-RO or OS-LFS and USER-LFS). By 2 MIPS, that emphasis has shifted so that the file system component is 19-21% and the location component is 15-17% Finally, by the disk-bound point, 3 MIPS, the location component is 0 (the USER-RO and OS-RO lines overlap, as do the USER-LFS, OS-LFS, and LFS-NOLOG) and the file system accounts for a 22% difference in performance. However, at any point along the curves, the best performance is provided by supporting transactions in the data manager on top of a log-structured file system.

As was observed in the disk-bound configuration, the size of the log does not contribute significantly to the performance of the systems. The difference in I/O costs between USER-LFS and OS-LFS is that USER-LFS is able to perform logical logging (proportional to the record size) while OS-LFS performs physical logging (proportional to page size). The logging difference between OS-LFS and LFS-NOLOG is that the LFS-NOLOG model requires a log even smaller than USER-LFS (16 bytes per modification rather than 2 records). Since logging is always performed at sequential speeds, the total time required to log a transaction is still a small part of the total I/O time (under 1%), and the resulting performance is the same for all three systems. Therefore, the primary benefit of the log-structured file system implementation is its superior write performance, not its ‘‘no-overwrite’’ policy.

4.5.3. Lock Contention

All the preceding tests were run with a uniform access pattern over the one gigabyte data file. The next issue to investigate is the effect of lock contention on these results. To induce contention, the database access pattern is skewed. The saturation point configuration for the disk-bound simulations has a multiprogramming level of 100, 10 disks, and 10 MIPS of available CPU. The distribution is varied from uniform (50/50; 50% of the accesses to 50% of the database) to extremely contention-bound (99/1; 99% of the accesses to 1% of the database). Figure 4-8 shows these results.

There are two factors at work here. First, since the configuration is initially disk-bound, the skewing of the access patterns results in a higher buffer cache hit ratio and therefore improved performance. Secondly, the skewing of the access patterns induces hot spots in the database, and the contention for locks degrades performance. At the 70/30 skew point, the USER-RO and OS-RO lines diverge as do the USER-LFS and OS-LFS/LFS-NOLOG lines. Since the user-level models use high concurrency locking on the indices, the user-level models continue to take advantage of the improved buffer cache hit ratio and their performance climbs steadily. The OS-RO model also exhibits improved performance, but not as much as the user-level systems since it is starting to suffer from contention on the indices, because index locks are held until transaction commit time in the embedded models. At the 80/20 point, the OS-LFS and LFS-NOLOG models actually suffer performance degradation as a result of the increased skewing and resulting contention. By the 90/10 point, the USER-LFS system has peaked and by the 95/5 point, all the models except the USER-RO have degraded dramatically.

Figure 4-9, which shows the number of aborts for each of the models as a function of this skewing, indicates that the embedded models exhibit higher abort rates than the user-level models from the 70/30 point until the 95/5 point. Since many more transactions are aborting, the resulting throughput is lower, therefore, in a contention-bound environment the coarse grain page locking employed by the embedded models is unsatisfactory.

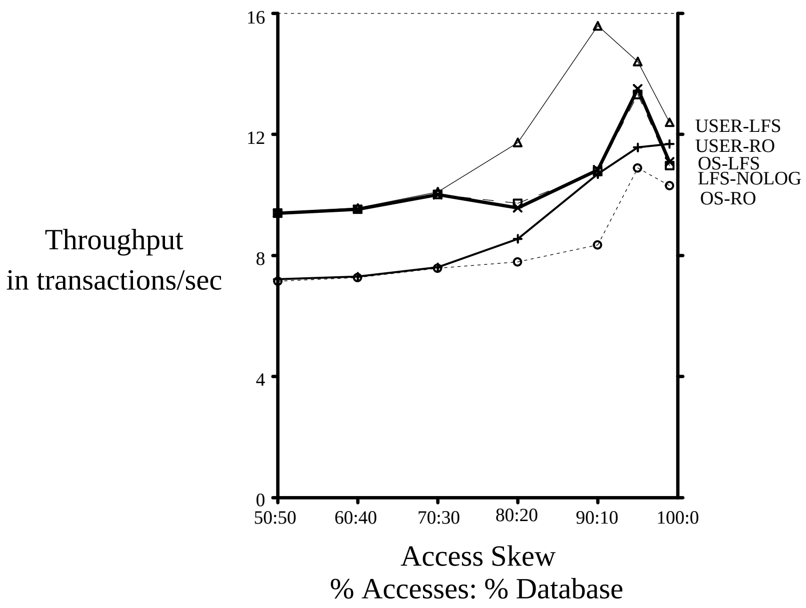


Figure 4-8: Effect of Skewed Access Distribution. Contention begins to impact performance when the skew reaches greater than 70/30. The embedded models diverge from their data manager counterparts at this point.

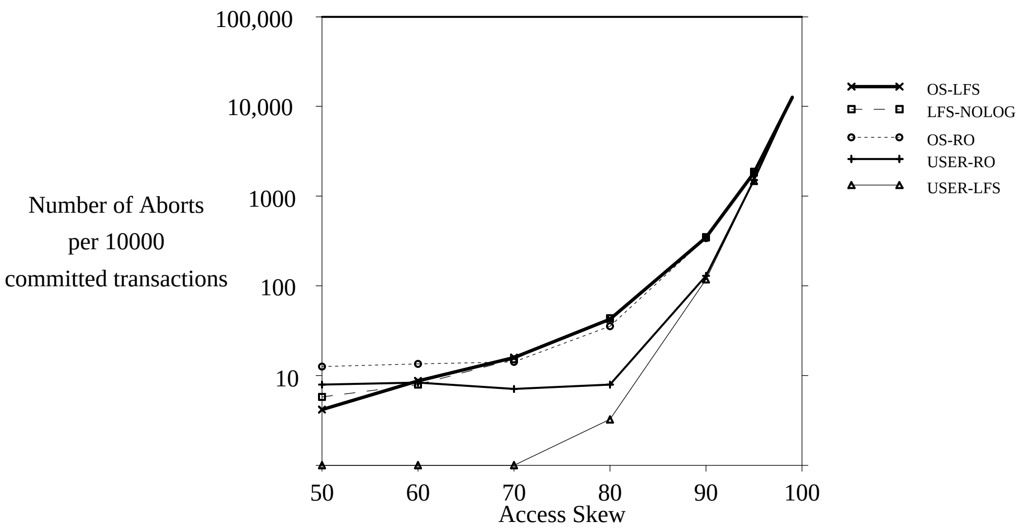


Figure 4-9: Effect of Access Skewing on Number of Aborted Transactions. The abort rate begins climbing at a 70/30 skew for the embedded systems, but at an 80/20 skew for the data manager.

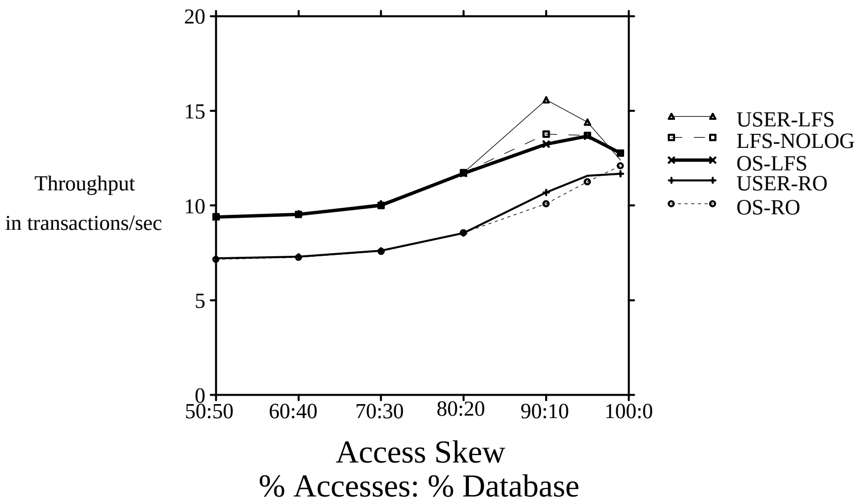


Figure 4-10: Effect of Access Skewing with Subpage Locking. By reducing the locking granularity, the embedded models can regain some of the performance lost to contention.

The next sections describe three techniques used to reduce the effect of lock contention in the embedded models. First, subpage locking, as described in section 4.4.2 was used. Next, the page size was reduced and locking was performed on full pages. Finally, a modified subpage locking technique similar to that described in [KUM89] was used.

Subpage locking reduces the locking granularity, and as a result, the degree of contention, but not as much as expected. Figure 4-10 shows the same contention-bound environment, shown in Figure 4-8, but uses subpage locking for the embedded models. In the region between 70/30 and 95/5 the embedded models come much closer to equaling their user-level counterparts. In the case of the read-optimized file systems (USER-RO and OS-RO), the difference is at most 6% (at the 90/10 point). For the log-structured file system, the largest gap is under 12% (also at 90/10). At the most contention-bound point the cause of contention moves from the indices to the data file and even the user-level models exhibit extreme contention. The reason that the embedded models exhibit better performance at the 99/1 point is because the user-level models continued to perform page locking on the data file. Obviously, the user-level models could use subpage locking in which case both user-level and embedded models would saturate at the same point.

Although subpage locking improved the throughput under high contention, the change was not as large as one might expect. Since updates to a B-Tree page require shuffling around the entries on a page, multiple subpages get locked for each update. The distribution of the additional pages which must be locked is skewed to favor subpages at the end of the page. This is shown in Figure 4-11. In addition, the CPU cost per level of the B-Tree is higher since multiple subpage locks are required to find the correct subpage. Therefore, if a high-contention environment is CPU-bound, changing the locking granularity will not improve performance. If the CPU is not the bottleneck, some of the performance lost to contention may be regained.

The next technique to reduce contention is to decrease the page size and lock full pages. While decreasing the page size reduces contention, it may also increase the depth of the B-Tree. Increasing the depth of the B-Tree may add extra I/O to each operation as well as adding an additional lock request to each traversal. As a result, reducing the page size is beneficial only if it does not increase the depth of the B-Tree. For the simulated database, reducing the page size

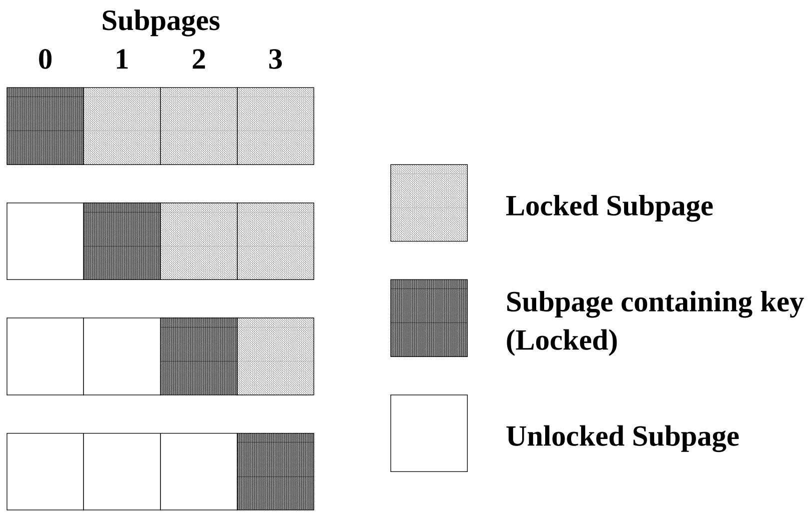


Figure 4-11: Distribution of Locked Subpages. Although subpage locking reduces the locking granularity, it does not dramatically reduce the probability of conflicts. Notice that subpage 3 is always locked if any key on the entire page is modified while subpage 1 gets locked only if it contains the modified key.

from 4 kilobytes to 2 kilobytes does not increase the depth of the B-Tree. The results in Figure 4-12 show the same contention-bound environment using page-locking and selecting the optimal page size for each model. The optimal page sizes were selected by simulating all page sizes between 128 bytes and 4 kilobytes and selecting the best one for each level of contention. The results in figure 4-12 used 4-kilobyte pages for skews of 50, 60, 70; 128 byte pages for 80, and 512 byte pages for 90, 95, and 99. Comparing these results to those shown in Figure 4-10 indicates that using page size to reduce contention is less effective than using subpage locking for the read-optimized file system. On the other hand, the embedded models on LFS perform much better with variable page sizes than with subpages. Furthermore, the write-optimized embedded models surpass the write-optimized user-level model at 95/5 rather than at 99/1 as before. Depending on the file system, varying either the subpage size or the page size is an effective mechanism for handling lock contention.

The last technique is the modified subpage locking. It is similar to the subpage locking described earlier, but it avoids the overhead of multiple locks per level of the B-Tree and the skewed distribution of the locked subpages. This is similar to the proposal in [KUM89], but has lower CPU costs. In both Kumar's algorithm and the one presented here, each subpage is treated as an independent bucket of entries. In Kumar's method, the smallest key for each subpage is stored on a page's first subpage. To locate a key, the first subpage is read-locked, the appropriate subpage is determined, and then that subpage is locked. Within each subpage, entries are chained in a linked list, requiring linear search time.

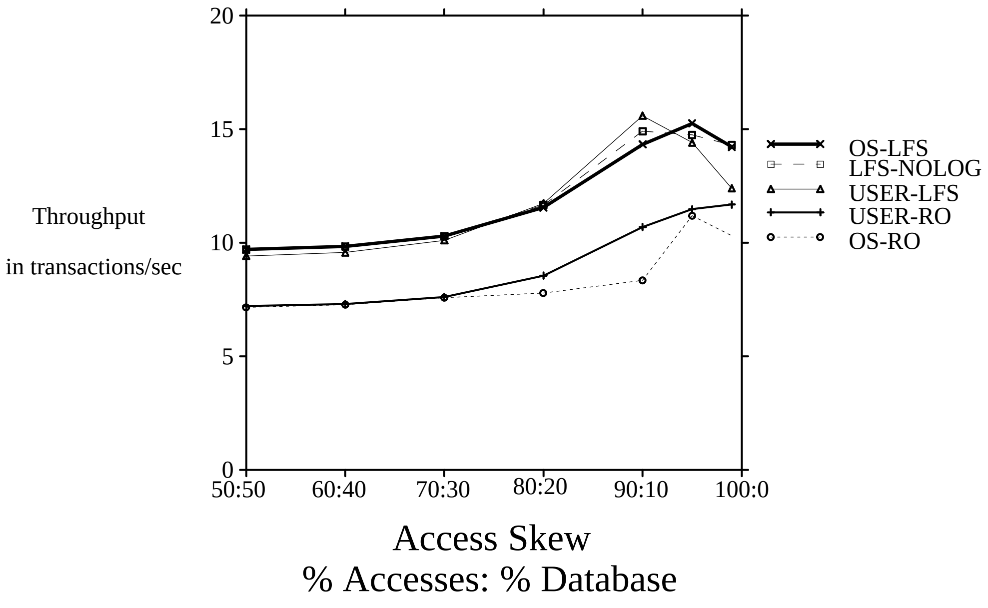


Figure 4-12: Effect of Access Skewing with Variable Page Sizes. In these tests, the embedded models perform comparably with the user-level models indicating that varying the page size compensates for some of the contention penalty in the embedded systems.

In the simulated algorithm, entries within a subpage are kept sorted maintaining the $O(log\ n)$ search time of a normal B-Tree whose page size is equal to the subpage size in our algorithm. The new algorithm avoids duplicating the key information on the first subpage and bottlenecking on that subpage by performing a non-locking binary search across subpages to locate the correct page. That page is then locked. To avoid conflicts between modifications to the low key on the page and the non-locking read, modifications to the low key on the page force a page reorganization. A page reorganization locks all the subpages and repartitions the keys across them. Therefore, when the non-locking read chooses a subpage, it will attempt to lock the subpage and fail. When the subpage lock becomes available, the waiter repeats the search and tries again. Page reorganization also occurs when a subpage fills.

While page reorganization may appear costly, the results in Figure 4-13 show this not to be the case. This simulation used modified subpage locking with a subpage size of 512 bytes and a page size of 4 kilobytes, yielding 22 keys per subpage on average.

During page reorganization, it is expected that half the entries on a page must be moved, so the reorganization is no more costly than a normal page-oriented delete. Whereas Kumar assumes that reorganization is required every 600 updates, this algorithm assumes reorganization is required once in every 10 updates since reorganization is required (and full page locking) both when subpages fill as well as when the first key on a page is modified.

Since this locking protocol offers the smaller locking granularity of subpage locking without the extra CPU overhead of multiple locks per update, its performance is even better than the data manager's performance, when the data file becomes the point of contention (since the data

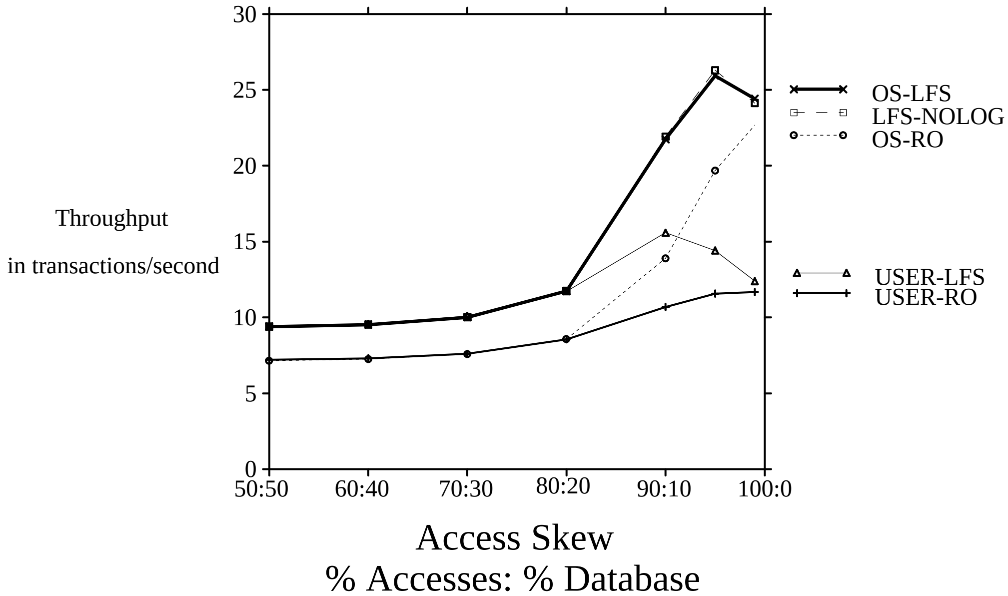


Figure 4-13: Effect of Access Skewing with Modified Subpage Locking. By reducing the locking granularity, the embedded systems are able to surpass the data manager in an environment with extremely high contention.

manager is still using page locking on the data file). Examining the number of aborts for the embedded models shown in Figure 4-14, the lock contention is virtually eliminated until the 90/10 point, and, beyond that point, the number of aborts in the embedded models is an order of magnitude smaller than for the data managers. Again, having the data manager use subpage locking on the data file in such high contention environments is clearly the right decision.

4.6. Conclusions

Independent of whether transaction support is embedded in the file system or implemented at user-level, the log-structured file system offers better performance than the traditional read-optimized file system. Its major benefit is its improved write performance, not its ‘‘no-overwrite’’ policy. In fact, as seen from the results in disk-bound configurations, the number of bytes logged has very little impact on the resulting performance. This is explained by the fact that logging always occurs at sequential speeds and is a very small fraction of the total I/O time.

Since logging is not an important factor embedded transaction support performs as well as the user-level support in disk-bound configurations. Whether a read-optimized or write-optimized file system is used, the user-level and embedded models offer nearly identical performance. As a result, supporting transactions within the file system is a feasible solution, when the system is disk-bound.

As Kumar concluded, when the CPU is the bottleneck, there is a penalty in embedding transaction support in a file system. However, when lock contention is not a factor, there is no need to perform subpage locking, and the difference in performance is directly proportional to the cost of

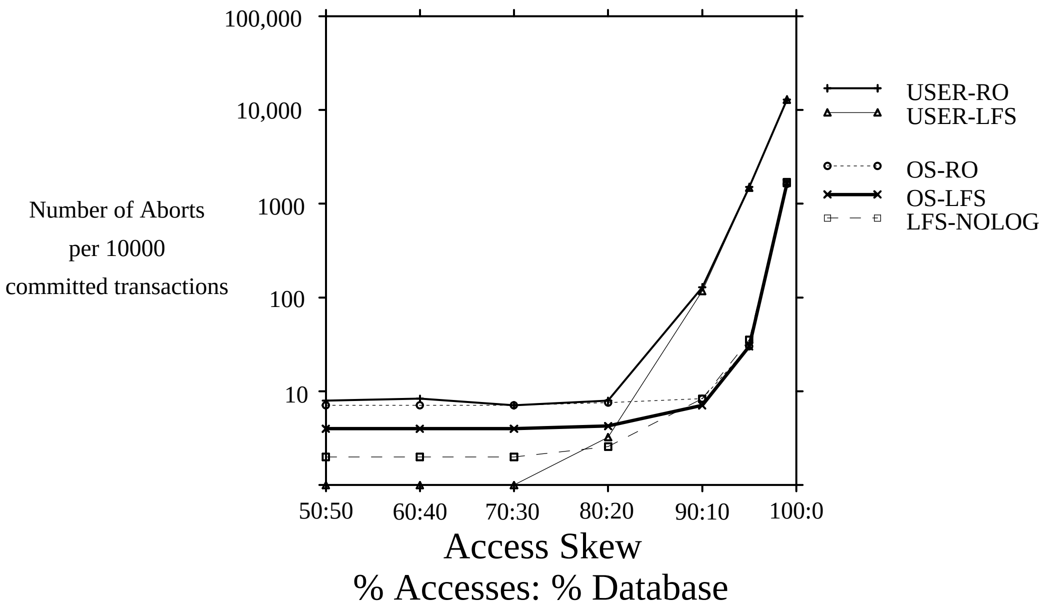


Figure 4-14: Effect of Modified Subpage Locking on the Number of Aborts. The new locking mechanism reduces the number of aborts by a factor of 10, thus allowing the high throughput rates observed in Figure 4-13.

a system call and is usually under 20%. Therefore, the feasibility of an embedded transaction manager is strictly dependent on the system call overhead.

Finally, as lock contention becomes a factor in limiting performance, all models experience some degradation, but the user-level system suffers the least due to its use of semantic information for B-Tree locking. The embedded models may recoup most of this performance loss through variable subpage and page sizes. In some cases, where the CPU is not a critical resource, embedded systems with modified subpage locking not only recoup this loss, but provide better throughput than the more traditional user-level architectures which perform page level locking on the data file. Obviously, the user-level models could use subpage locking as well, with the expectation that both models would perform comparably.

Except in the most CPU-bound environments, there is virtually no penalty incurred in embedding transaction support in the operating system. It does, however, require careful and defensive design to avoid index contention as well as operating system flexibility to vary the page and subpage sizes as needed.

There are several areas which warrant further investigation. These simulations did not take into account the cost of cleaning (garbage collection) in the log-structured file system. This will reduce the benefit of the log-structured file system and will be examined by means of implementation in the following chapter. However, the use of RAID devices [PATT88] will penalize the small writes that occur in a read-optimized file system and make the log-structured file system appear more desirable.

Chapter 5

Transaction Support in a Log-Structured File System

The simulation study described in Chapter 4 compared the performance of a transaction application in both user-level and embedded implementations using both a log-structured and a read-optimized file system. According to that simulation the log-structured file system offered better performance than the read-optimized file system for a short-transaction workload, and the embedded transaction manager performed as well as a user-level transaction manager except in highly contentious environments. In this chapter, empirical results will be presented. These results will account for overhead introduced by the cleaner and highlight some inaccuracies or exaggerations in the simulation results.

The performance analysis here focuses on two points. First the transaction manager embedded in LFS is compared to a more conventional transaction architecture (i.e. one implemented as a user-level process). Then the user-level transaction system on LFS is compared to the same user-level system on a more conventional file system.

5.1. A User-Level Transaction System

For the experiments described in this chapter, a traditional transaction system using write-ahead logging (WAL) and two-phase locking [GRAY76] was implemented. The implementation platform was a DECstation 5000 running the Sprite operating system [OUST88]. The Sprite application programming interface is largely UNIX compatible. This user-level system is used as a basis for comparison to LFS-embedded support. The next sections discuss the design tradeoffs and module architecture of the user-level implementation.

5.1.1. Crash Recovery

The recovery protocol is responsible for providing transaction semantics. There are a wide range of recovery protocols available [HAER83], but they can roughly be divided into two major categories. The first category records all modifications to the database in a separate log file, and uses the log to back out or reapply modifications if a transaction aborts or the system crashes. The second category avoids the use of a log by carefully controlling when data are written to disk. The former can be called the *logging protocols* and the latter the *non-logging protocols*.

Non-logging protocols retain dirty buffers in main memory or temporary files until transaction commit, forcing these pages to disk at that time. During a long-running transaction, temporary files can be used to hold dirty pages that may need to be evicted from memory before commit, but in the Sprite environment, the only user-level mechanism to force pages from the buffer cache to disk is the *fsync*(2) system call. Unfortunately, *fsync*(2) is an expensive system call in that it forces all the pages of a file to disk, and the application must issue one of these system calls per file.

In addition, *fsync*(2) provides no way to control the order in which dirty pages are written to disk. Since non-logging protocols must sometimes order writes carefully [SULL92], they are difficult to implement on UNIX systems. As a result, a logging protocol was chosen.

Logging protocols can be categorized based on how information is logged (physically or logically) and how much is logged (before-images, after-images or both). In *physical logging*, images of complete physical units (pages or buffers) are recorded, while in *logical logging* a description of the operation is recorded. Therefore, while entire pages are recorded in a physical log, only the records being modified are recorded in a logical log. In fact, physical logging can be thought of as a special case of logical logging, since the ‘‘records’’ that are logged in logical logging might be physical pages. Since logical logging is both more space-efficient and more general, it was selected.

In *before-image logging*, a copy of the original data is logged, while in *after-image logging*, the new data is logged. If only before-images are logged, then there is sufficient information in the log to allow transaction *undo* (go back to the state represented by the before-image). However, if the system crashes and a committed transaction’s changes have not reached the disk, there is insufficient information to *redo* the transaction (reapply the updates). Therefore, logging only before-images necessitates forcing dirty pages at commit time. As mentioned above, forcing pages at commit is quite costly.

If only after-images are logged, then there is sufficient information in the log to allow transaction *redo* (go forward to the state represented by the after-image), but there is not enough information required to *undo* transactions that aborted after dirty pages were written to disk. Therefore, logging only after-images necessitates holding all dirty buffers in main memory until commit or writing them to a temporary file.

Since neither constraint (forcing pages on commit or buffering pages until commit) was feasible, both before- and after-images were logged. To ensure that log records are available for any data pages that need to be redone or undone, *write-ahead logging* (WAL) is used. In WAL, the log is written to disk before any of the data it describes are written to disk. This means that the only file that ever needs to be forced to disk is the log. Since the log is append-only, modified pages always appear at the end and may be written to disk efficiently in any file system that favors sequential ordering (e.g., the fast file system, a log-structured file system, or an extent-based system).

5.1.2. Concurrency Control

The concurrency control protocol is responsible for maintaining consistency in the presence of concurrent accesses. There are several alternative solutions such as locking, optimistic concurrency control [KUNG81], and timestamp ordering [BERN80]. Since optimistic methods and timestamp ordering are generally more complex and restrict concurrency without eliminating starvation or deadlocks, *two-phase locking* (2PL) was used. In strict 2PL all locking occurs in two phases. In the first phase, locks are acquired for all accessed data. In the second phase, locks may be released. Once phase two has begun, no further locks may be requested, so most systems perform phase two at transaction commit. Strict 2PL is suboptimal for certain data structures, (e.g. B-Trees) because it can limit concurrency, so a special locking protocol based on one described in [LEHM81] is used.

5.1.3. Management of Shared Data

In order to provide concurrent data access and enforce write-ahead logging described in Section 5.1.1 a shared-memory buffer manager is included. Not only does this provide the guarantees required for WAL, but a user-level buffer manager is frequently faster than using the file system buffer cache [STON81]. Reads or writes involving the file system buffer cache often require copying data between user and kernel space while a user-level buffer manager can return pointers to data pages directly. When multiple processes require the same page, all processes access the same page in a shared-memory buffer pool while using the operating system buffer cache usually requires each process to make a local copy.

5.1.4. Module Architecture

The preceding sections described a set of algorithms for managing the transaction log, locks, and a cache of shared buffers. Figure 5-1 shows the main interfaces and architecture of the user-level implementation. An application is constructed by linking in a library containing each of the modules depicted in Figure 5-1 and explained in detail below.

5.1.4.1. The Log Manager

The Log Manager enforces write-ahead logging. Its primitive operations are *log*, *log_commit*, *log_read*, *log_roll*, and *log_unroll*. The *log* call performs a buffered write of the specified log record and returns a unique log sequence number (LSN). The LSN can be used to retrieve a record from the log using the *log_read* call. The *log* interface knows very little about the internal format of the log records it receives. Rather, all log records are referenced by a header structure, a log record type, and a character buffer containing the data to be logged. The log record type is used to call the appropriate redo and undo routines during abort and commit processing. While the Log Manager is used to provide before- and after-image logging, it may also be used to implement any of the logging algorithms described earlier.

The *log_commit* operation behaves exactly like the *log* operation but guarantees that the log has been forced to disk-before returning. Group commit [DEWI84] is used to reduce the per-transaction commit cost. The point at which commit processing actually occurs is determined by three thresholds. The first is the *group threshold* and defines the minimum number of transactions that must be active in the system before group commit happens. The second is the *wait*

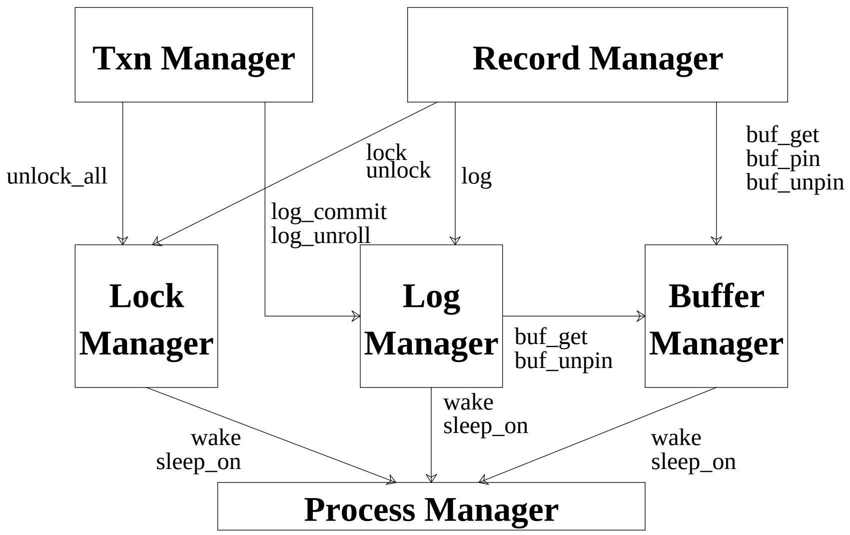


Figure 5-1: Library Module Interfaces.

threshold, expressed as the percentage of active transactions waiting to be committed. The last is the *logdelay threshold* that indicates how many dirty log pages should be allowed to accumulate before a waiting transaction's commit record is flushed.

Log_unroll reads log records from the log, following backward transaction pointers and calling the appropriate undo routines to implement transaction abort. In a similar manner, *log_roll* reads log records sequentially forward, calling the appropriate redo routines to recover committed transactions after a system crash. It is called from the recovery program.

5.1.4.2. The Buffer Manager

The Buffer Manager uses a pool of shared memory to provide a least-recently-used (LRU) page cache. Transactions request pages from the buffer manager and keep them *pinned* to ensure that they are not written to disk while they are in a logically inconsistent state. When page replacement is necessary, the Buffer Manager finds an unpinned page and then checks with the Log Manager to ensure that write-ahead logging is enforced.

5.1.4.3. The Lock Manager

The Lock Manager supports general purpose locking (single writer, multiple readers), which is currently used to provide two-phase locking and high concurrency B-Tree locking. However, the general purpose nature of the lock manager provides the ability to support a variety of locking protocols. Currently, all locks are issued at the granularity of a page (the size of a buffer in the buffer pool), which is identified by two 4-byte integers (a file id and page number). This provides the necessary information to extend the Lock Manager to perform hierarchical locking [GRAY76].

If an incoming lock request cannot be granted, the requesting process is queued for the lock and descheduled. When a lock is released, the wait queue is traversed and any newly available locks are granted. Locks are located via a hash table and are chained by both object and transaction. The transaction chains facilitate rapid traversal of the lock table during transaction commit and abort.

The primary interfaces to the lock manager are *lock*, *unlock*, and *lock_unlock_all*. *Lock* obtains a new lock for a specific object. There are also two variants of the *lock* request, *lock_upgrade* and *lock_downgrade* that allow the caller to atomically trade a lock of one type for a lock of another. *Unlock* releases a specific mode of lock on a specific object. *Lock_unlock_all* releases all the locks associated with a specific transaction.

If multiple transactions are active concurrently, deadlocks can occur and must be detected and resolved. A user-level process, the deadlock detector, monitors the lock table checking for deadlocks. When a deadlock is found, the deadlock detector randomly selects one of the deadlocked transactions and aborts it.

5.1.4.4. The Process Manager

The Process Manager acts as a user-level scheduler to make processes wait on unavailable locks and pending buffer cache I/O. For each process, a semaphore is maintained upon which that process waits when it needs to be descheduled. When a process needs to be run, its semaphore is cleared, and the operating system reschedules it. No sophisticated scheduling algorithm is applied; if the lock for which a process was waiting becomes available, the process is made runnable.

5.1.4.5. The Transaction Manager

The Transaction Manager provides the standard interface of *txn_begin*, *txn_commit*, and *txn_abort*. It keeps track of all active transactions, assigns unique transaction identifiers, and directs the abort and commit processing. When a *txn_begin* is issued, the Transaction Manager assigns the next available transaction identifier, allocates a per-process transaction structure in shared memory, increments the count of active transactions, and returns the new transaction identifier to the calling process. The in-memory transaction structure contains a pointer into the lock table for locks held by this transaction, the last log sequence number, a transaction state (*idle*, *running*, *aborting*, or *committing)*, an error code, and a semaphore identifier.

At commit, the Transaction Manager calls *log_commit* to record the end of the transaction and to flush the log. It then directs the Lock Manager to release all locks associated with the given transaction. If a transaction aborts, the Transaction Manager calls *log_unroll* to read the transaction's log records and undo any modifications to the database. As in the commit case, it then calls *lock_unlock_all* to release the transaction's locks.

5.1.4.6. The Record Manager

The Record Manager supports the abstraction of reading and writing records to a database. The database access routines **dbopen**(3) [BSD91] have been modified to call the log, lock, and buffer managers. In order to provide functionality to perform undo and redo, the Record Manager defines a collection of log record types and the associated undo and redo routines. The Log Manager performs a table lookup on the record type to call the appropriate routines. For example, the B-Tree access method requires two log record types: insert and delete. A replace operation is implemented as a delete followed by an insert and is logged accordingly.

5.2. The Embedded Implementation

In the embedded model, transaction support is implemented within the file system. As suggested in [MITC82], transaction-protection can be assigned to files selectively. A simple utility provides the user with the ability to turn transaction protection on and off on a per-file basis. The interface to transaction-protected files is identical to the interface to unprotected files (*open*, *close*, *read*, and *write*). Four new system calls, *txn_begin*, *txn_abort*, *txn_commit*, and *txn_detect*, complete the interface.

The system calls perform similar functionality to their user-level counterparts. When *txn_begin* is called, a transaction identifier is assigned to the requesting process and a data structure that describes the state of the transaction is initialized. *Txn_abort* discards all modified pages associated with the transaction, releases all locks held by the transaction, and marks the transaction state as aborted. *Txn_commit* forces modified buffers for the transaction to disk, releases the transaction's locks and marks the transaction state as committed. The last system call, *txn_detect*, performs deadlock detection, detecting cycles of processes waiting on each other for locks and aborting one of the deadlocked transactions.

When transactions are embedded in the file system, the need for many of the modules presented in the user-level implementation disappears. The operating system's buffer cache replaces the user-level buffer cache and the kernel scheduler obviates the need for any user-level process management. No explicit logging is performed, but the ‘‘no-overwrite’’ policy observed by LFS guarantees the existence of before-images, and requiring that all dirty transaction pages be written to disk at commit (the FORCE commit policy of [HAER83]) guarantees the existence of after-images in the file system. Therefore, the only functionality that needs to be added to the kernel is lock and transaction management. Figures 5-2 and 5-3 show the two architectures. The embedded system in 5-3 eliminates much of the redundancy present in Figure 5-2, producing a simpler architecture. In this architecture, multiple data management facilities could use the same kernel mechanisms rather than implementing their own. In terms of lines of code, the embedded implementation added approximately 1200 lines of code to the operating system but removed 10,000 lines of code from the user-level data manager.

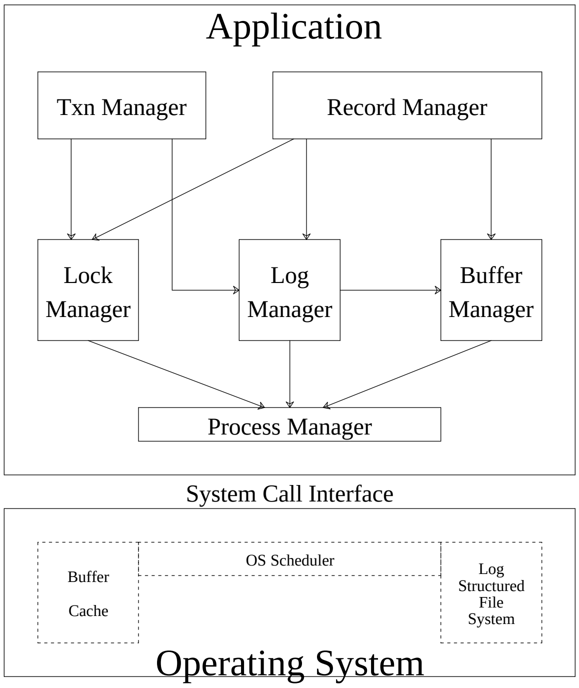


Figure 5-2: User-Level System Architectures. The user-level library duplicates much of the functionality already present in the operating system, including logging, buffer management, and scheduling.

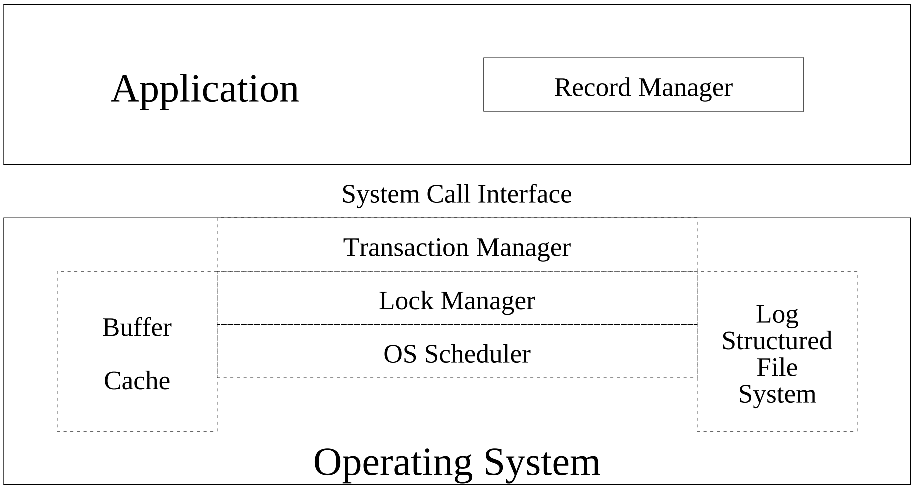


Figure 5-3: Embedded Transaction System Architecture. The user-level buffer manager has been replaced by the operating system's buffer cache. The log manager is replaced by LFS, and the user-level process module has been replaced by the operating system's scheduler. The lock management module and the transaction module have been moved into the operating system.

5.2.1. Data Structures and Modifications

The operating system required two new data structures and modification to three existing data structures in order to support embedded transactions. The new structures are the *lock table* and the *transaction state*. The structures requiring modification are the *inode*, *file system state*, and the *process state*. Each is described below.

5.2.1.1. The Lock Table

The lock table maintains a hash table of currently locked objects, identified by file and logical block number. It is very similar to the user-level lock manager data structures described in Section 5.1.4.3. The lock table is an operating system global data structure. Locks are chained by both object and transaction, facilitating rapid traversal of the lock table during transaction commit and abort. Figure 5-4 depicts the organization of the lock table.

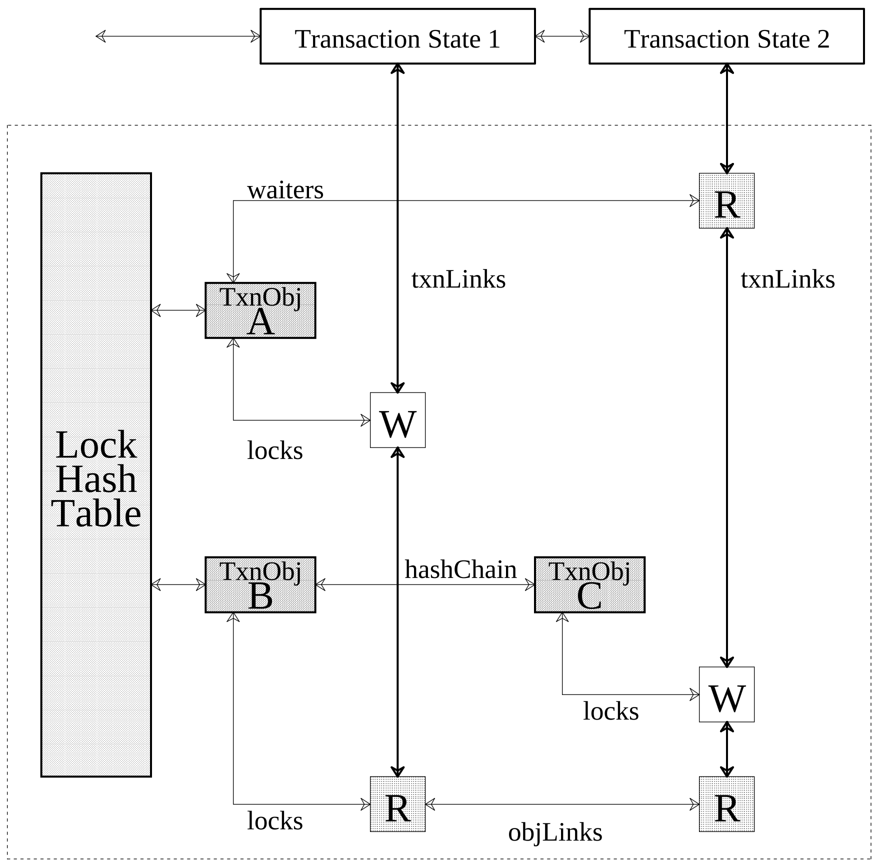


Figure 5-4: The Operating System Lock Table. This picture shows two transactions labeled 1 and 2. Transaction 1 holds 2 locks, a write lock on object A and a read lock on Object B. Transaction 2 holds 2 locks, a read lock on object B and a write lock on C. It is also waiting for a read lock on Object A.

5.2.1.2. The Transaction State

The transaction state is a per-transaction structure, similar to the user-level one discussed in Section 5.1.4.5. It contains the status of the transaction (*idle*, *running*, *aborting*, *committing*), a pointer to the chain of locks the transaction holds, a transaction identifier, and links to other transaction states. This structure is linked to the process state and is depicted in Figure 5-4.

5.2.1.3. The Inode

The inode structure is the in-memory and on-disk structure that describes the disk layout of the file. It contains the physical representation of the index structure described in Section 4.1. The inode and indirect block structures are depicted in Figure 5-5. In addition to the block information, the inode contains file attribute information like size, time of last access, time of last modification, ownership, permissions, etc. The in-memory representation of the inode additionally includes lists of buffers and links to other inodes. The transaction implementation extends the on-disk inode to include information that indicates if the file is transaction-protected. The in-memory inode is extended to have a list of transaction-protected buffers in addition to its clean and dirty buffer lists.

5.2.1.4. The File System State

The file system state is an in-memory data structure that describes the current state of the file system. It is extended to contain a pointer to the transaction lock table so that all transaction locks for a file system are accessible from a single point (as opposed to from each process with an

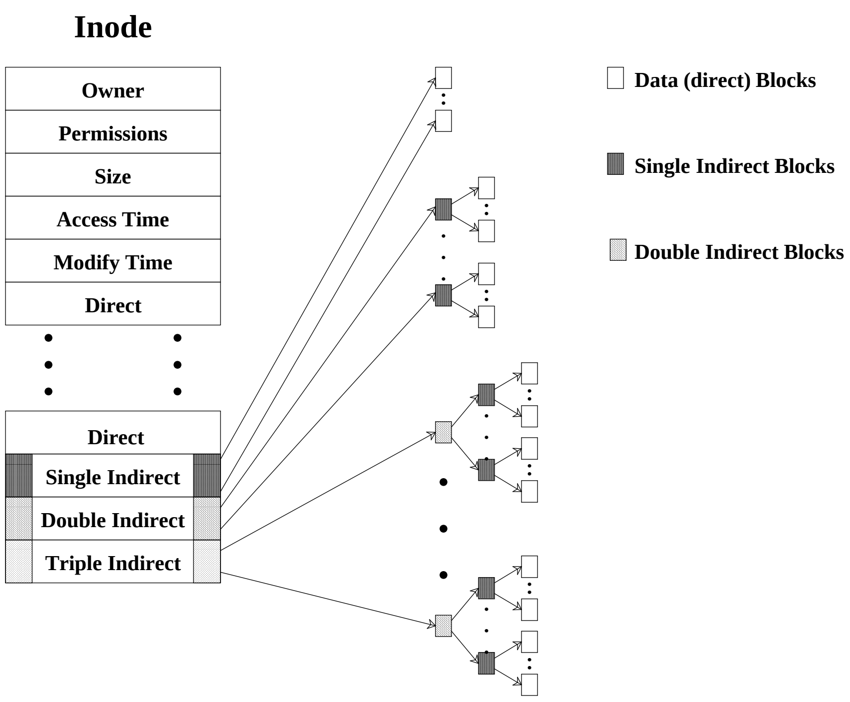


Figure 5-5: File Index Structure (inode).

active transaction).

5.2.1.5. The Process State

The process state maintains information about all currently active processes. It contains links to run and sleep queues and to other active processes. In addition, it records the process permissions, resource limits and usage, the process identifier, and a list of open files for the process. It is extended to include a pointer to the transaction state.

5.2.2. Modifications to the Buffer Cache

The *read* and *write* calls behave nearly identically to those in the original operating system. A read request is specified by a byte offset and length. This request is translated into one or more page requests serviced through the operating system's buffer cache. If the blocks belong to a transaction-protected file, a read lock is requested for each page before the page request is issued (either from the buffer cache or by reading it from disk). If the lock can be granted, the read continues normally. If it cannot be granted, the process is descheduled and left sleeping until the lock is released. Writes are implemented similarly, except that a write lock is requested instead of a read lock.

5.2.3. The Kernel Transaction Module

Where the user-level model provides a subroutine interface, the embedded model supports a system call interface for transaction begin, commit, and abort processing. At *txn_begin*, a new transaction structure is created if the process has never had a transaction, or it is initialized if the process has an existing transaction structure. The next available transaction identifier maintained by the operating system is assigned, and the transaction's lock list is initialized.

When a process issues a *txn_abort*, the kernel locates the lock chain for the transaction through the transaction state. It then traverses the lock chain, releasing locks and invalidating any dirty buffers associated with those locks. Transactions may also be aborted by the deadlock detector. The deadlock detector runs periodically, and when it finds a deadlock, one of the participating transactions is aborted. The aborted transaction's state is modified to reflect that it was aborted by the deadlock detector. Since any transaction involved in a deadlock was, by definition, waiting on a lock, it was either performing a read or write to the file system. The read or write of the aborted transaction will return an error value and set the error number to a special value, indicating an aborted transaction.

At *txn_commit*, the kernel traverses the transaction's lock chain, and flushes dirty buffers, referenced by the locks, to disk. Once all the dirty buffers have been flushed, the kernel releases locks. In the case where only part of a page is modified, the entire page is still written to disk at commit. This compares badly with logging schemes where only the updated bytes need be written. While it might be expected that the increased amount of data flushed at commit results in a heavy performance penalty, the simulation results in Chapter 4 indicate that this is not true. Rather, the overall transaction time is so dominated by random reads to databases too large to cache in main memory that the additional sequential bytes written during commit have no impact on the resulting performance. Furthermore, forcing dirty blocks at commit obviates the need to write these blocks later when their buffers need to be reclaimed.

5.2.4. Group Commit

Since the kernel implementation uses a FORCE policy, every transaction causes one or more disk writes at commit. In the same way that database systems use group commit to amortize the cost of flushing the log [DEWI84], LFS uses group commit to amortize the cost of flushing blocks. Rather than flushing a transaction's blocks when it issues a *txn_commit*, the process sleeps until a timeout interval has elapsed or until sufficiently more transactions have committed to justify the write. If the embedded implementation uses the same group commit policy as a user-level implementation, it should produce the same performance improvement.

5.2.5. Implementation Restrictions

Section 5.2.2 makes no mention of protecting a file's meta-data. Indirect blocks do not pose a special problem in that they are updated only when data pages are written to disk, which happens only at commit. Therefore, only the addresses of committed pages are reflected in indirect blocks. On the other hand, inodes must be handled differently since inodes are updated frequently, and a single inode describes an entire file, so locking the inode would, in effect, lock the entire file. During a read, the *access time* is updated, while during a write both the *access time* and *modification time* are updated. In addition, if a write extends a file, the *size* of the file may change. There are two ways in which a file's length may change: the file may add bytes to or delete bytes from the end of an existing block or it may allocate or deallocate a block.

In the first case, the changes are localized to a single block and the write lock on that block is sufficient to prevent any data corruption. However, if transactions are allowed to read the *size* field from the inode, inconsistent results are possible:

Time	Transaction 1	Transaction 2
1	lock last block	
2	append bytes	
3	update size	
4		read size
5	abort	

In this example, transaction 2 read the size of the file after transaction 1 had changed it. Then transaction 1 aborted, making the value read by transaction 2 invalid. Even if the modification of the *size* field is delayed, there is the potential for violation of serializability.

Time	Transaction 1	Transaction 2
1	update A	
2	lock last block	
3	append bytes	
4		read size
5	update size	
6	commit	
7		use size and A

In this example, the serializability violation is manifested by transaction 2's value of A being the value after transaction 1 committed, but its value of *size* being the value before transaction 1 committed.

The case in which a block is allocated or deallocated is even more troublesome than the previous examples. Consider two transactions each trying to extend the file. If transaction 1 extends the file and locks and writes block N, then transaction 2 can lock and write block N+1. Now, what happens if transaction 1 aborts? The possibilities are:

- Leave unallocated blocks in the file (i.e. do not undo appends).
- Do not allow multiple appenders.

Since the benchmarks presented here use formatted files (B-Trees and fixed-length record database files), leaving unallocated blocks (holes) in files is an adequate solution, and that is the strategy implemented. When an extend operation occurs, the block is immediately added to the file. If the transaction aborts, the allocated page becomes a hole in the file. A more versatile solution is desired and systems like Quicksilver [HASK88] discuss various ways of handling similar situations.

The next four sections address the other areas where the current implementation is deficient, namely:

- All dirty buffers must be held in memory until commit ([HAER83] NO-STEAL semantics).
- Locking is strictly two-phase and is performed at the granularity of a page.
- Transactions may not span processes.
- Processes have only one transaction active at any point in time.
- It is not obvious how to do media recovery.

For the benchmark described in Section 5.3.1 (a modified TPC-B), the NO-STEAL policy is sufficient since transactions are small and short-lived. With regard to page locking, the simulation study in Chapter 4 indicated that locking at granularities smaller than a page is required only for highly contentious environments.

5.2.5.1. Support for Long-Running Transactions

In the implementation described, buffers belonging to uncommitted transactions cannot be evicted from the buffer cache. To support STEAL semantics (allowing uncommitted pages to be evicted from the buffer pool), the algorithm that writes dirty buffers to disk must be modified to call a special *txn_flush* routine that evicts these pages.

The protocol for writing these pages is similar to the shadow page protocol used in System R [ASTR76]. A shadow file must be assigned to each transaction file with uncommitted pages that are written to disk. The shadow file's identifier is recorded in the inode of the transaction file and the dirty buffers are written to the shadow file. In this way, both the old copies (before the transaction) and the new copies (during the transaction) simultaneously exist. At *txn_commit*, when buffers would normally be scheduled for writing, the transaction file's meta-data is updated to contain the block in the shadow file and the shadow file's meta-data is updated to no longer contain the block. Rather than having to flush all the dirty buffers at commit, only the remaining dirty buffers and the two inodes need be flushed. If the system crashes, all shadow files can be removed since it is known that their pages belong to uncommitted transactions. Shadow paging is sometimes found objectionable because it destroys clustering within a file [CHAM81]. Since LFS is writing the shadow pages to the same place as it would normally write the file pages, these objections are not applicable.

The cleaner also needs to be modified to handle shadow files correctly. During typical operation, the cleaner reads a segment and uses the information in the segment summary to determine if each block is still ‘‘live’’. For each block, the cleaner looks at its inode and finds the disk address corresponding to the block in question. If the disk address is the same as that being cleaned, then the block is active, otherwise it is ‘‘dead’’ and can be cleaned. However, in the current cleaner, if the block formerly belonged to a shadow file and has since been committed and moved to the transaction file, the check will fail and the cleaner will consider the block ‘‘dead’’. Instead, shadow files must be recognizable to the cleaner, and the cleaner must compare block addresses against both the shadow and the original file. The block is ‘‘dead’’ only if it belongs to neither file.

5.2.5.2. Support for Subpage Locking

The challenge in subpage locking is to permit one transaction to modify and commit a subpage while another transaction is modifying another subpage on the same page. To accomplish this, the functionality of shadow files must be extended to incorporate shadow pages. Unlike the implementation described in the last section, shadow files must be kept per transaction.[5] When a page is read into the cache, it is assigned a version number, initialized to 0. When a transaction modifies a subpage, a shadow copy of the page is created with the version number of the original page, the subpage of the shadow page is modified, and the shadow page is entered into the transaction's shadow file. At commit, the shadow page is *reconciled* (made consistent) with the original page, the original page's version number is incremented, and the original page is flushed to disk. Then the shadow pages and files are freed.

Page reconciliation is the action of making original pages consistent with committing shadow pages. First, when a transaction creates a shadow page, it always copies the original page in the buffer cache, not a shadow. At commit, the committing transaction compares its shadow page version number to the version number of the current original. If no other transactions with modifications on this page have committed, the version numbers will be the same, and the committing transaction's shadow is flushed to disk, becomes the original, and has its version number incremented, and the old original is freed. If the version numbers do not agree, then the subpages modified by the committing transaction are copied to the current original, which is then flushed to disk. The current original's version number is incremented, and the shadow page is freed. This is the algorithm simulated in Chapter 4.

5.2.5.3. Support for Nested Transactions and Transaction Sharing

The embedded model implemented and analyzed in this chapter was designed to avoid changing the existing kernel interface. If transaction identifiers are added to the *read* and *write* system calls, then nested transactions, concurrent active transactions within the same process, and transactions spanning multiple processes (*i.e. transaction sharing*) could be implemented. Transaction sharing is particularly useful if system utilities (such as compilers, assemblers, and source code control systems) want to transaction-protect sets of operations.

Transaction sharing can be accomplished by adding two system calls, *txn_transfer* and *txn_continue*. The *txn_transfer* call allows a process with an active transaction to pass control for the transaction to another process. The kernel moves the transaction structure from the transferring process to the target process. The target process takes control of the transaction by issuing the *txn_continue* call that checks for a transferred process and waits if one does not yet exist. If no *txn_continue* is made for the transferred transaction, the transaction is either aborted or returned to the transferring process. There are a myriad of possibilities for orchestrating the transfer of transactions. Some examples are:

- The transferring process waits until the transaction has been accepted.
- The transferring process is signaled if the transaction is not accepted in a defined interval.
- The continuing process signals another process to request a transfer.

5.2.5.4. Support for Recovery from Media Failure

Since there is no separate log in this implementation, it is not obvious how to support recovery from media failures. Using mirrored disks or RAID techniques [PATT88] to improve reliability is one option, but this still does not address catastrophic failures (earthquakes, fires, etc.). Distributed disk arrays offer a solution for this problem [SCHL90]. In a conventional logging system, the protection against catastrophic failure involves sending log records to a remote site. The distributed RAID technique creates a physical log record of the bit differences in a modified block. As in a conventional system, these log records are dispatched to a remote site. However, rather than merely storing the log at the remote site, the log record is used to compute

[5] A performance optimization allows multiple transactions to share shadow files until updates by different transactions request subpages that are part of the same page.

parity across a distributed array. In the case of a site failure, the remaining sites in the array have enough information to reconstruct the database at the failed site.

5.3. Performance

The performance measurements reported here are from a DECstation 5000/200[6] running the Sprite operating system. The system has 32 megabytes of memory and a 300 megabyte RZ55 SCSI disk drive. The database resides on the 300 megabyte disk while binaries were served from remote machines, accessed via the Sprite distributed file system. Reported times are the means of five tests and have standard deviations within two percent of the mean.

The performance analysis is divided into three sections: transaction performance, non-transaction performance, and sequential read performance. The transaction benchmark compares the LFS embedded system with the conventional, user-level system on both the log-structured and read-optimized file systems. The non-transaction benchmark is used to show that kernel transaction support does not impact non-transaction applications. The sequential read test measures the impact of LFS' write-optimized policy on the sequential read performance.

5.3.1. Transaction Performance

To evaluate transaction performance, a modified version of the industry standard TPC-B transaction processing benchmark [TPCB90] was used. The TPC-B benchmark simulates a withdrawal performed by a hypothetical teller at a hypothetical bank. The database consists of relations (files) for accounts, branches, tellers, and history. For each transaction, the account, teller, and branch balances must be updated to reflect the withdrawal, and a history record is written that contains the account id, branch id, teller id, and the amount of the withdrawal. The account, branch, and teller relations were all implemented as primary B-Tree indices (the data resides in the B-Tree file) while the history relation was implemented as a fixed-length record file, providing access sequentially and by record number. The test database was configured for a 10 transaction per second (TPS) system according to the TPC-B scaling rules:

1 branch / TPS
10 tellers / TPS
100,000 accounts / TPS

The implementation of the benchmark differs from the specification in three aspects. First, the specification requires that the database keep redundant logs on different devices, but only a single log was used. Second, all tests were run on a single, centralized system, so there was no notion of remote accesses. Third, the tests were run single-user (multiprogramming level of one), providing a worst-case analysis. The configuration measured is so disk-bound that increasing the multi-programming level increases throughput only marginally. See [SELT92] for a detailed discussion of the performance impact in a multi-user environment.

In this test, three systems were evaluated: a user-level transaction manager on a traditional operating system, a user-level transaction manager on a log-structured file system, and a transaction manager embedded in a log-structured file system. The two interesting comparisons are comparing the user-level transaction manager on a log-structured file system to a user-level transaction manager on a traditional file system and comparing the user-level transaction manager on LFS to the kernel level transaction manager in LFS.

Figure 5-6 shows the results of this test. As expected, the LFS system outperformed the conventional file system, but by a disappointing 10%. The fact that there was a difference at all is because LFS flushes dirty pages from the buffer pool more efficiently. When the user process flushes a page, the page is cached in the kernel's buffer pool, and eventually flushed to disk. In the LFS case, this write occurs as part of a segment write and takes place at near-sequential

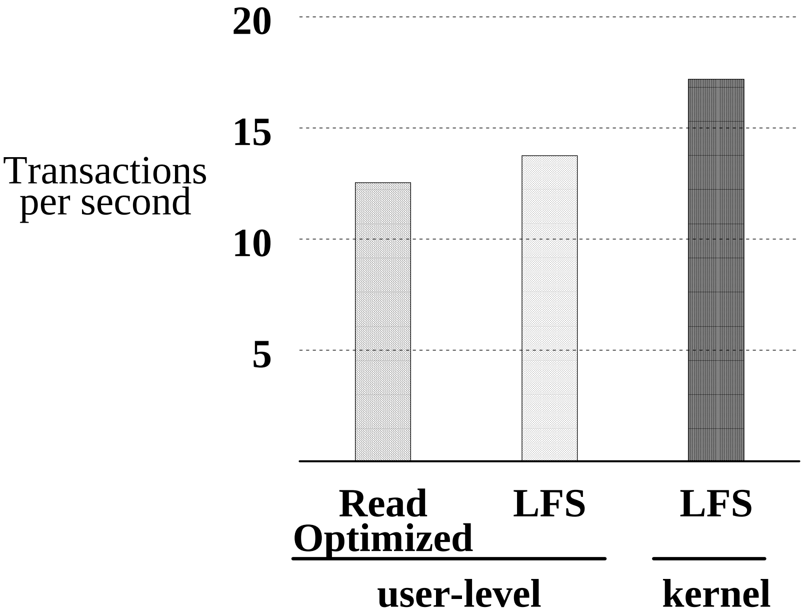


Figure 5-6: Transaction Performance Summary. The leftmost two bars compare performance of a user-level transaction manager on the original Sprite file system (read-optimized) and on LFS. The rightmost two bars compare the performance of the user-level transaction manager on LFS to the LFS embedded transaction manager.

speed. In the read-optimized case, this write occurs within 30 seconds of when it entered the buffer cache and is sorted in the disk queue with all other I/O to the same device (the random reads). Thus, the overhead is greater than the overhead of the sequential write in LFS.

The 10% performance improvement discussed above (12.3 TPS v.s. 13.6 TPS) is disappointing when compared to the disk-bound simulation in Chapter 4, which predicted a 27% performance improvement. The difference between the implementation and the simulation is explained by two things. First, a log-structured file system requires the presence of a cleaner, a garbage collection process that reclaims space in the file system resulting from deleted or overwritten blocks. While the simulation did not take into account any cleaner overhead, in practice, the cleaner substantially disrupts processing.

When the cleaner runs, it locks out all accesses to the particular files being cleaned. In this benchmark there are only four data files being accessed, so these files are also the ones being cleaned. Therefore, when the cleaner locks these files, no other processing can occur. As a result, there is a noticeable ‘‘hiccup’’ in performance. The benchmark prints out a message each time it processes 100 transactions and the cleaner prints out a message when it begins and finishes cleaning. The benchmark prints out its messages at a steady rate until the cleaner prints out its begin message. There are no messages from the benchmark for the 30-60 seconds before the cleaner prints out its finish message. Once the cleaner completes, the benchmark resumes at its original rate. In a commercial environment, this disturbance in performance is obviously unacceptable.

[6] DEC is a trademark of Digital Equipment Corporation.

The second reason for the difference between the simulation and the implementation is that the simulation ignores much of the system overhead, focusing on the transaction processing operations. For example, the simulation does not account for query processing overhead, context switch times, system calls other than those required for locking, or process scheduling. As a result, the total transaction time is much greater and the difference in performance is a smaller percentage of the total transaction time.

The next comparison contrasts the performance of the user-level and kernel implementations on LFS. Once again, the simulation results in Chapter 4, predicting no difference between user-level and kernel models, differ from implementation results. A fundamental assumption made in the simulation was that synchronization would be much faster in the user-level model than in the kernel model. The argument was that user-level processes could synchronize in shared memory without involving the kernel while synchronization in the kernel model required a system call. Unfortunately, the test platform, the DECstation 5000, does not have a hardware test-and-set instruction. As a result, the user-level model used system calls to obtain and release semaphores, doubling the synchronization overhead of the kernel implementation that required a single system call. This synchronization overhead exactly accounts for the difference between the user and kernel implementations [SELT92]. Techniques described in [BERS92] describe how to implement user synchronization quickly on a system without hardware test-and-set eliminating the performance gap shown in Figure 5-6.

5.3.2. Non-Transaction Performance

This test was designed to run programs that do not use the embedded transaction manager to determine if its existence in the kernel affects the performance of applications that do not use it. This test used three applications. The first is the user-level transaction system since it does not take advantage of any of the new kernel mechanisms. The second, Andrew [HOWA88], is an engineering workstation file system test. It consists of copying a collection of small files, creating a directory structure, traversing the directory hierarchy, and performing a series of compiles. The third, ‘‘Bigfile’’, was designed to measure throughput of large file transfers. It creates, copies, and removes a set of 10-20 relatively large files (1 megabyte, 5 megabytes, and 10 megabytes on a 300 megabyte file system).

The goal of this test was to demonstrate that adding transactions to the operating system did not impact the performance of applications that did not take advantage of the embedded support. All three benchmarks were run on the unmodified operating system and on the one with embedded transaction support. Since the transaction code is isolated from the rest of the system, no difference in performance was expected. The results are summarized in Figure 5-7 and show that there is virtually no impact for any of the tests. In all tests, the difference between the two systems was within 1-2% of the total elapsed time and within the standard deviations of the test runs. This is the expected result, as non-transaction applications pay only a few instructions in buffer access determining that transaction locks are unnecessary.

5.3.3. Sequential Read Performance

The improved write performance of LFS is not without its costs. The log-structured file system optimizes random writes at the expense of future sequential reads. To construct a worse case test for LFS, begin with a sequentially written file, randomly update the entire file, and then read the file sequentially. The SCAN test consists of the final sequential read phase and was designed to quantify the penalty paid by sequentially reading a file after it has been randomly updated. After creating a new account file, 100,000 TPC-B transactions are executed. The account file is approximately 160 megabytes or 40,000 4-kilobyte pages, and the 100,000 transactions should touch a large fraction of these pages, leaving the database randomly strewn about the disk. Then the file is sequentially read in key order.

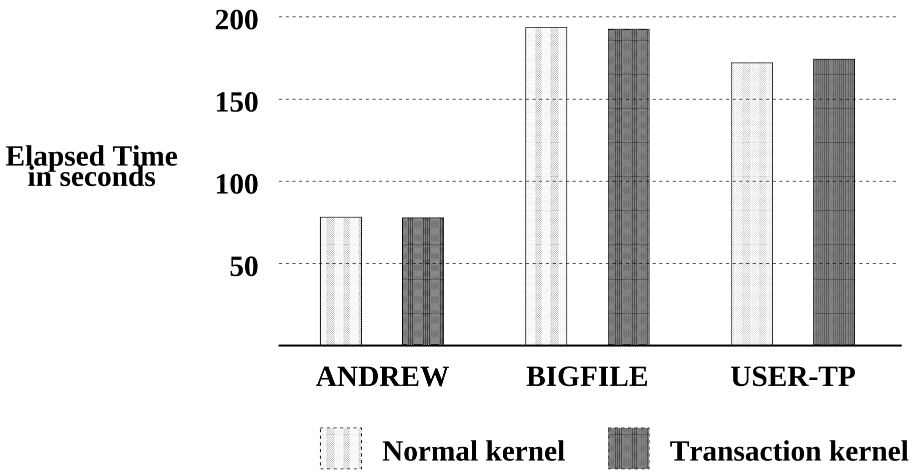


Figure 5-7: Performance Impact of Kernel Transaction Support. None of the three benchmarks used the kernel transaction support. As is shown by the similarity in elapsed times for all benchmarks, the embedded support did not decrease the overall system performance.

Figure 5-8 shows the elapsed time for the SCAN test. As expected, the traditional file system (read-optimized) significantly outperforms LFS. The conventional file system paid disk seek penalties during transaction processing to favor sequential layout. As a result, it demonstrates 33% better performance than LFS during the sequential test.[7]

There are two ways to interpret this result. The first, naive approach says that you get a small (10%) improvement in the transaction workload, but you pay a large (50%) penalty in the sequential workload. However, a more complete interpretation considers the two workloads (transactional and sequential) together. The time gained by write-optimization during the transaction workload can be directly traded off against the time lost during the sequential read. This tradeoff can be quantified by calculating how many transactions must be executed between sequential reads so that the total elapsed time for both file systems is the same.

In Figure 5-9, the total elapsed time for both the transaction run and the sequential run is given as a function of the number of transactions executed before the sequential processing. This is actually a pessimistic result for LFS. The degradation in sequential performance observed by LFS is a function of the number of transactions that have been executed prior to the sequential scan. For example, if only a single transaction is run before initiating the sequential scan, the structure of the database will be largely unchanged and the sequential read time for LFS would be nearly identical to that for the read-optimized system. However, the data in the graph shows the total elapsed time for transactions and sequential scan assuming that the sequential scan always takes as long as it did after 100,000 transactions. Even so, the point at which the two lines intersect is approximately 134,300 transactions and represents how many transactions need to be executed, per scan, to realize a benefit from the log-structured file system. From the perspective of time, at 13.6 TPS, the system would have to run for approximately 2 hours 40 minutes to reach the crossover point. That is, if the transaction workload runs at peak throughout for less than 2

[7] This test does not correspond exactly to reading the raw file sequentially since the file is read in key order.

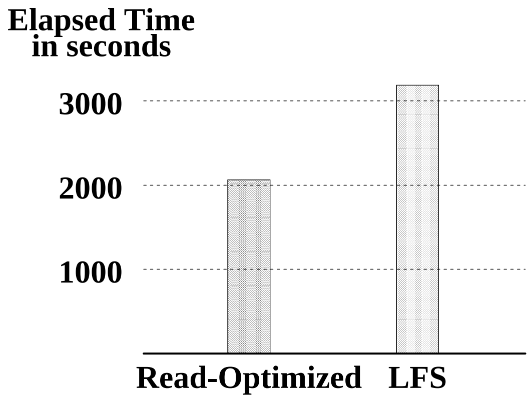


Figure 5-8: Sequential Performance after Random I/O. This graph shows the elapsed time to scan the entire account file after 100,000 TPC-B transactions were executed. The file system that favors sequential layout (read-optimized) was approximately 50% faster than LFS. Note that since the metric was elapsed time, higher numbers indicate worse performance.

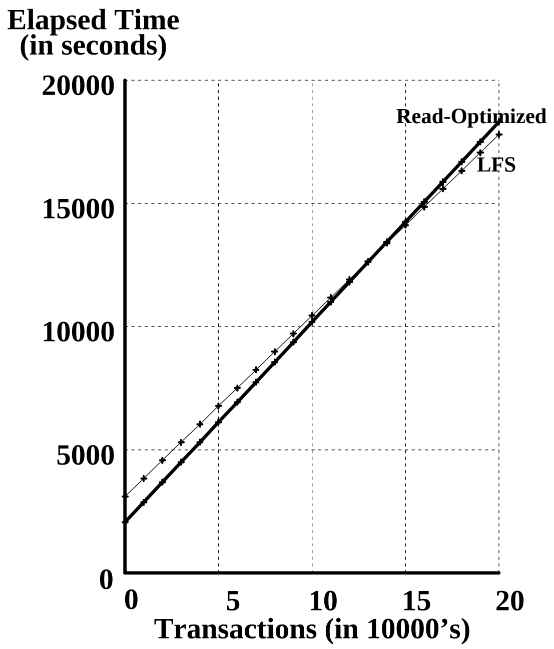


Figure 5-9: Elapsed Time for Combined Benchmark. The results here show the total elapsed time for both the transaction processing workload and the sequential batch run. Applications that require sequential processing after some period of transaction processing will observe better overall performance from the read-optimized system when the number of transactions executed is less than 134,300 and from LFS when more than that number are executed.

hours 40 minutes before a sequential pass is made, the read-optimized system is providing better overall performance, but if the transaction workload runs for longer than that, LFS provides the better overall performance. The ratio of transactions to sequential runs will be extremely workload dependent. In an automatic teller environment, short transactions are executed nearly 24 hours per day, while sequential scans occur very infrequently. However, in data-mining applications, the majority of the processing is more likely to be complex query processing with infrequent transactional updates.

This is not an entirely satisfying result. In practice, LFS needs to address the issue of sequential read access after random write access. Since LFS already has a mechanism for rearranging the file system (the cleaner), this mechanism might well be used to coalesce files that become fragmented. This goal, in part, drove the redesign of LFS discussed in Chapter 6.

5.4. Conclusions

In this chapter, it has been shown that a log-structured file system has the potential to improve the performance of transaction processing applications. Currently, LFS provides a 10% performance improvement over a conventional file system on a modified TPC-B workload. Although one can construct workloads where an embedded model doesn't perform as well as user-level models, the embedded system does look viable, providing performance comparable to that of the user-level system. Such an implementation enables applications to easily incorporate transaction-protection with only minor modification. Products such as source code control systems, software development environments (e.g. combined assemblers, compilers, debuggers), and system utilities (user registration, backups, ''undelete'', etc.), as well as database systems could take advantage of this additional file system functionality. However, sequential read performance after random write performance still poses a problem for LFS. The next chapter will address this issue.

Chapter 6

Redesigning LFS

The previous chapter highlighted some shortcomings in the original LFS design. Specifically, the disruption in service due to cleaning and the poor sequential read performance are issues of grave concern. Although this chapter will discuss many modifications to LFS, the modification that addresses the problems raised in Chapter 5 is moving the cleaner into user-space and providing it with four system calls to communicate with the kernel. This accomplishes two goals.

First, it becomes easy to prevent the cleaner from locking out other applications while it is running. Synchronization between the cleaner and the kernel occurs during a system call where cleaned blocks are checked against recently modified blocks to make sure that newer blocks are not overwritten. This alleviates the disruption in processing observed during the TPC-B benchmark.

Secondly, moving the cleaner into user-space makes it simple to experiment with different cleaning policies and implement multiple cleaners with different policies. By allowing a variety of cleaning policies, the sequential read performance penalty that occurs after random updates can potentially be reduced or eliminated. This chapter presents a new design of LFS which addresses these issues.

The rest of this chapter is organized as follows. Section 6.1 describes the detailed design of a log-structured file system, contrasting its disk layout and recovery to that of the Fast File System. Section 6.2 discusses the major issues that drove the redesign, contrasting the decisions made with those found in the original LFS. Section 6.3 highlights some of the implementation issues unique to the new implementation and integration with the fast file system, and Section 6.4 concludes the chapter. Throughout the remainder of this chapter, the original LFS implementation is referred to as Sprite-LFS while the new implementation is referred to as BSD-LFS, as it is part of the 4.4BSD release from the Computer Systems Research Group at the University of California, Berkeley.

6.1. A Detailed Description of LFS

There are two fundamental differences between an LFS and a traditional UNIX file system, as represented by the Fast File System (FFS) [MCKU84]: the on-disk layout of the data structures and the recovery model. While Section 4.1 presented a very high-level description of a log-structured file system, this section describes the key structural elements of an LFS, contrasting the data structures and recovery to FFS. The complete design and implementation of Sprite-LFS can be found in [ROSE92]. Table 6-1 compares key differences between FFS and LFS. The reasons for these differences will be described in detail in the following sections.

6.1.1. Disk Layout

In both FFS and LFS, a file's physical disk layout is described by an index structure (*inode*) that contains the disk addresses of some *direct*, *indirect*, *doubly indirect*, and *triply indirect blocks*. Direct blocks contain data, while indirect blocks contain disk addresses of direct blocks, doubly indirect blocks contain disk addresses of indirect blocks, and triply indirect blocks contain disk addresses of doubly indirect blocks. For the remainder of this chapter, inodes and indirect

Task	FFS	LFS
Assign disk addresses	block creation	segment write
Allocate inodes	fixed locations	appended to log
Maximum number of inodes	statically determined	grows dynamically
Map inode numbers to disk addresses	static address	lookup in inode map
Maintain free space	bit maps	cleaner segment usage table
Make file system state consistent	fsck	roll-forward
Verify directory structure	fsck	background checker

Table 6-1 Comparison of File System Characteristics of FFS and LFS

blocks are referred to as meta-data.

The FFS is described by a *superblock* that contains file system parameters (block size, fragment size, and file system size) and disk parameters (rotational delay, number of sectors per track, and number of cylinders). The superblock is replicated throughout the file system to allow recovery from crashes that corrupt the primary copy of the superblock. The disk is statically partitioned into cylinder groups, typically between 16 and 32 cylinders to a group. Each group contains a fixed number of inodes (usually one inode for every two kilobytes in the group) and bit maps to record inodes and data blocks available for allocation. The inodes in a cylinder group reside at fixed disk addresses, so that disk addresses may be computed from inode numbers. New blocks are allocated to optimize for sequential file access. Ideally, logically sequential blocks of a file are allocated so that no seek is required between two consecutive accesses. Because data blocks for a file are typically accessed together, the FFS policy routines try to place data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. Figure 6-1 depicts the physical layout of FFS.

LFS is a hybrid between a sequential database log and FFS. It performs all writes sequentially, like a database log, but incorporates the FFS index structures into this log to support efficient random retrieval. In an LFS, the disk is statically partitioned into fixed-size segments, typically one-half megabyte. The logical ordering of these segments creates a single, continuous log.

An LFS is described by a superblock similar to the one used by FFS. When writing, LFS gathers many dirty pages and prepares to write them to disk sequentially in the next available segment. At that time, LFS sorts the blocks by logical block number, assigns them disk addresses, and updates the meta-data to reflect their addresses. The updated meta-data blocks are gathered with the data blocks, and all are written to a segment. As a result, the inodes are no longer in fixed locations, so, LFS requires an additional data structure, called the *inode map* [ROSE90], that maps inode numbers to disk addresses.

Since LFS writes dirty data blocks into the next available segment, modified blocks are written to the disk in different locations than the original blocks. This space reallocation is called a ‘‘no-overwrite’’ policy, and it necessitates a mechanism to reclaim space resulting from deleted or overwritten blocks. The *cleaner* is a garbage collection process that reclaims space from the file system by reading a segment, discarding ‘‘dead’’ blocks (blocks that belong to deleted files or that have been superseded by newer blocks), and appending any ‘‘live’’ blocks. In order for the cleaner to determine which blocks in a segment are ‘‘live,’’ it must be able to identify each block

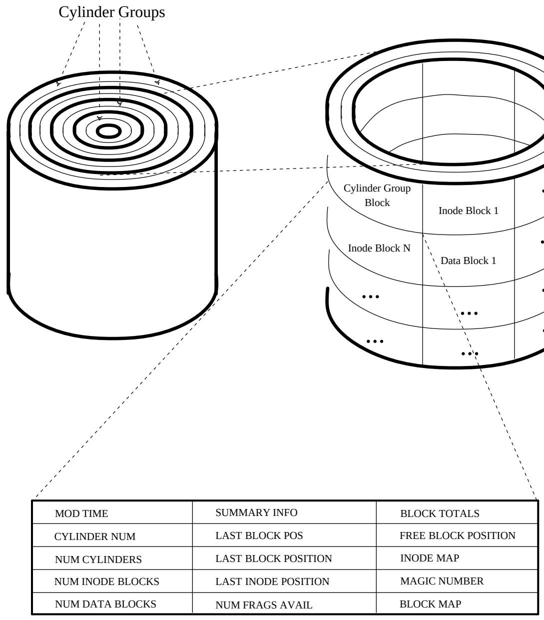


Figure 6-1: Physical Disk Layout of the Fast File System. The disk is statically partitioned into cylinder groups, each of which is described by a cylinder group block, analogous to a file system superblock. Each cylinder group contains a copy of the superblock and allocation information for the inodes and blocks within that group.

in a segment. This determination is done by including a *summary block* in each segment that identifies the inode and logical block number of every block in the segment. In addition, the kernel maintains a *segment usage table* that shows the number of ‘‘live’’ bytes and the last modified time of each segment. The cleaner uses this table to determine which segments to clean [ROSE90]. Figure 6-2 shows the physical layout of LFS.

While FFS flushes individual blocks and files on demand, LFS must gather data into segments. Usually, there will not be enough dirty blocks to fill a complete segment [BAKER92], in which case LFS writes *partial segments*. A physical segment contains one or more partial segments. For the remainder of this thesis, *segment* will be used to refer to the physical partitioning of the disk, and *partial segment* will be used to refer to a unit of writing. Small partial segments most commonly result from NFS operations or *fsync*(2) requests, while writes resulting from the *sync*(2) system call or system memory shortages typically form larger partials, ideally taking up an entire segment. During a *sync*, the inode map and segment usage table are also written to disk, creating a *checkpoint* that provides a stable point from which the file system can be recovered in case of system failure.

6.1.2. File System Recovery

There are two aspects to file system recovery: bringing the file system to a physically consistent state and verifying the logical structure of the file system. When FFS or LFS add a block to a file, there are several different pieces of information that may be modified: the block itself, the inode, the free block map, indirect blocks, and the location of the last allocation. If the

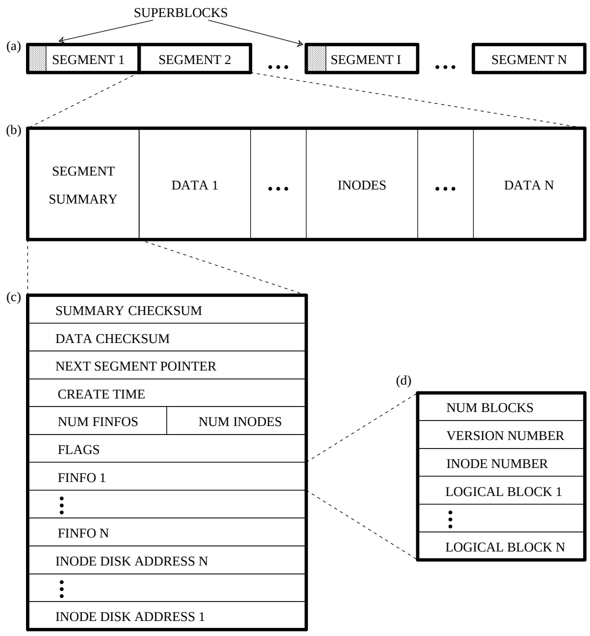


Figure 6-2: Physical Disk Layout of a Log-Structured File System. A file system is composed of *segments* as shown in Figure (a). Each segment consists of a summary block followed by data blocks and inode blocks (b). The *segment summary* contains checksums to validate both the segment summary and the data blocks, a timestamp, a pointer to the next segment, and information that describes each file and inode that appears in the segment (c). Files are described by *FINFO* structures that identify the inode number and version of the file (as well as each block of that file) located in the segment (d).

system crashes during the addition, the file system is likely be left in a physically inconsistent state. There is currently no way for FFS to localize inconsistencies. As a result, FFS must rebuild the entire file system state, including cylinder group bit maps and meta-data. At the same time, FFS verifies the directory structure and all block pointers within the file system. Traditionally, *fsck*(8) is the agent that performs both of these functions.

In contrast to FFS, LFS writes only to the end of the log and is able to locate potential inconsistencies and recover to a consistent physical state quickly. This part of recovery in LFS is more similar to standard database recovery [HAER83] than to *fsck*. It consists of two parts: initializing all the file system structures from the most recent checkpoint and then ‘‘rolling forward’’ to incorporate any modifications that occurred subsequently. The roll forward phase consists of reading each segment after the checkpoint in time order and updating the file system state to reflect the contents of the segment. The next segment pointers in the segment summary facilitate reading from the last checkpoint to the end of the log, the checksums are used to identify valid segments, and the timestamps are used to distinguish the partial segments written after the checkpoint and those written before which have been reclaimed. The file and block numbers in the FINFO structures are used to update the inode map, segment usage table, and inodes, making the blocks in the partial segment extant. As is the case for database recovery, the recovery time is proportional to the interval between file system checkpoints.

Phase I	Traverse inodes Validate all block pointers. Record inode state (allocated or unallocated) and file type for each inode. Record inode numbers and block addresses of all directories.
Phase II	Sort directories by disk address order. Traverse directories in disk address order. Validate ‘‘.’’. Record ‘‘..’’. Validate directories’ contents, type, and link counts. Recursively verify ‘‘..’’.
Phase III	Attach any unresolved ‘‘..’’ trees to lost+found. Mark all inodes in those trees as ‘‘found’’.
Phase IV	Put any inodes that are not ‘‘found’’ in lost+found. Verify link counts for every file.
Phase V	Update bit maps in cylinder groups.

Table 6-2: Five Phases of *fsck.*

While standard LFS recovery quickly brings the file system to a physically consistent state, it does not provide the same guarantees made by *fsck*. When *fsck* completes, not only is the file system in a consistent state, but the directory structure has been verified as well. The five passes of *fsck* are summarized in Table 6-2. For LFS to provide the same level of robustness as FFS, LFS must periodically make many of the same checks. While LFS has no bit maps to rebuild, the verification of block pointers and directory structure and contents is crucial for the system to recover from media failure. This recovery will be discussed in more detail in Section 6.2.4.

6.2. Design Issues

The Sprite implementation of LFS succeeded in its goal of dramatically improving write performance, however it had several deficiencies that made it unsuitable for a production environment. The major concerns with Sprite-LFS that this redesign addresses are as follows:

1. Sprite-LFS consumes excessive amounts of memory.
2. Write requests are successful even if there is insufficient disk space.
3. Recovery does nothing to verify the consistency of the file system directory structure.
4. Segment validation is hardware dependent.
5. All file systems use a single cleaner and a single cleaning policy.
6. There are no performance numbers that measure the cleaner overhead.

The description of LFS in Section 6.1 focused on the overall strategy of log-structured file systems. The following sections discuss how BSD-LFS addresses the first five problems listed above. Section 6.3 addresses the implementation issues specific to integration in a BSD framework. The last issue is addressed in Chapter 7.

In most ways, the logical framework of Sprite-LFS is unchanged. The segmented log structure and the major support structures associated with the log, namely the inode map, segment usage table, and cleaner remain. However, to address the problems described above and to integrate LFS into a BSD system, nearly all of the details of implementation, including a few fundamental design decisions have been altered. Table 6-3 summarizes the design differences between Sprite-LFS and BSD-LFS indicating the reason for the change. The following sections describe the major design issues in more detail.

Design Point	Sprite-LFS	BSD-LFS	Reason
Memory Consumption	Reserves large quantity of physical memory	Makes no memory reservations	Tying down multiple megabytes of physical memory was unacceptable in a production environment.
Block Accounting	Block accounting is performed when blocks are written to disk.	Block accounting is performed when blocks are written into the cache.	The file system cannot accept writes into the cache if it does not have the physical disk space to which to write the data.
Segment Validation	Presence of summary block validates segment.	Checksum (of data) in summary block validates segment.	The file system cannot assume that disks write data in the order presented.
File System Verification	Assumes file system is consistent after LFS roll-forward.	Verifies directory structure in the background after roll-forward.	Media corruption or faulty hardware can result in corruption of the disk in places other than the location of the last write.
The Cleaner	Runs as kernel process.	Runs as a user process.	Placing the cleaner in user-space allows for multiple cleaners and experimentation with different cleaning policies, and prevents the cleaner from locking out writers.
Inode Map and Segment Usage Table	Maintained as special kernel data structures.	Maintained in regular file (ifile).	Making these structures part of a regular file allows the cleaner to easily read them and reduces the amount of special purpose code in the kernel.
Directory Operations	Maintains on-disk log.	Provides atomic updates across segments.	Since the file system is a log, it seemed wrong to have an additional log inside it.
File Access Times	Stored in inode map.	Stored in inode.	The access time is 8 bytes and would have increased the size of the inode map by 67%. Since the inode map should be cached to achieve best performance, this seemed like a bad tradeoff.
Inode Allocation	Inodes are allocated sparsely with clustering by directory.	Free inodes are chained in a free list.	The sparse nature of the inode map makes it larger than desirable and if directories get large, many entries must be traversed to find a free inode.
Superblock	Single superblock.	Superblock replicated throughout the file system.	If the superblock is corrupted, it is nearly impossible to reconstruct the file system.

Table 6-3: Design Changes Between Sprite-LFS and BSD-LFS.

6.2.1. Memory Consumption

Sprite-LFS assumes that the system has a large physical memory and reserves substantial portions of it. The following storage is reserved:

Two 64-kilobyte or 128-kilobyte staging buffers
Since not all devices support scatter/gather I/O, data is written in buffers large enough to allow the maximum transfer size supported by the disk controller, typically 64 kilobytes or 128 kilobytes. These buffers are allocated per file system from kernel memory.

One cleaning segment
One segment's worth of buffer cache blocks per file system are reserved for cleaning.

Two read-only segments
Two segments' worth of buffer cache blocks per file system are marked read-only so that they may be reclaimed by Sprite-LFS without requiring an I/O.

Buffers reserved for the cleaner
Each file system also reserves some buffers for the cleaner. The number of buffers is specified in the superblock and is set during file system creation. It specifies the minimum number of unmodified buffers that must be present in the cache at any point in time. On the Sprite cluster, the total amount of buffer space reserved for the cleaner on 10 commonly used file systems was 37 megabytes.

One segment
This segment (typically one-half megabyte) is allocated from kernel memory for use by the cleaner. Since this one segment is allocated per system, only one file system per system may be cleaned at a time.

The reserved memory described above makes Sprite-LFS a very ‘‘bad neighbor’’ as kernel subsystems compete for memory. While memory continues to become cheaper, a typical laptop system has only three to eight megabytes of memory, and might very reasonably expect to have three or more file systems.

BSD-LFS greatly reduces the memory consumption of LFS. First, BSD-LFS does not use separate buffers for writing large transfers to disk, instead it uses the regular buffer cache blocks. For disk controllers that do not coalesce contiguous operations, 64-kilobyte staging buffers (briefly allocated from the kernel memory pool) are used for the transfers. The size of the staging buffer was set to the minimum of the maximum transfer sizes for currently supported disks. However, simulation results in [CAR92] show that for current disks, the write size that minimizes the read response time is typically about two tracks; two tracks is close to 64 kilobytes for the disks on our systems.

Secondly, moving the cleaner into user-space avoids the need to reserve a large amount of main memory in the kernel. Instead, the user-level process competes for virtual memory space with the other processes.

Third, rather than reserving two read-only segments per file system, BSD-LFS keeps track of how many dirty buffers it has accumulated and begins a segment write before memory becomes a critical resource. When the number of buffers dirtied by LFS exceeds the *start write threshold*, a segment write, which should generate more clean buffers, is initiated. In the meantime, if the number of dirty buffers exceeds the *stop access threshold*, any LFS read and write requests for buffers not currently in the cache will wait. The difference between the total number of buffers headers in the system and the *stop access threshold* is analagous to Sprite-LFS's read-only buffers. However, in BSD-LFS, while these buffers are not available to LFS, they are available to the virtual memory system and other file systems (such as the memory-based file system [MCKU90]).

6.2.2. Block Accounting

Sprite-LFS maintains a count of the number of disk blocks available for writing (i.e. the real number of disk blocks that do not contain useful data). This count is decremented when blocks are actually written to disk. This approach implies that blocks can be successfully written to the cache but can fail to be written to disk if the disk becomes full before the blocks are actually written. Even if the disk is not full, all available blocks may reside in uncleaned segments and new data cannot be written. To prevent the system from deadlocking or losing data in these cases, BSD-LFS uses two forms of accounting.

The first form of block accounting is similar to that maintained by Sprite-LFS. BSD-LFS maintains a count of the number of disk blocks that do not contain useful data. It is decremented whenever a new block is created in the cache. Since many files die in the cache [BAKER91], this number is incremented whenever blocks are deleted, even if they were never written to disk.

The second form of accounting keeps track of how many blocks are available in clean segments and have not been allocated for dirty buffers present in the cache. This space is allocated as soon as a dirty block enters the cache, but is not reclaimed until segments are cleaned. This count is used to initiate cleaning. If an application attempts to write data and there is no space currently available for writing, the write will sleep until space is available. These two forms of accounting guarantee that if the operating system accepts a write request from the user, barring a crash, the data will be written.

Accounting for the actual disk space available is difficult because inodes are not written into dirty buffers and segment summaries aren’t created until the segment is written. Every time a clean inode is updated in the inode cache, a count of inodes to be written is incremented. When blocks are dirtied, the number of available disk blocks is decremented. To decide if there is enough disk space to allow another write into the cache, the number of segment summaries necessary to write what is in the cache is computed, added to the number of inode blocks necessary to write the dirty inodes and compared to the amount of space available on the disk. To create more available disk space, either the cleaner must run or dirty blocks in the cache must be deleted.

6.2.3. Segment Structure and Validation

Sprite-LFS places segment summary blocks at the end of partial segments trusting that if the write containing the segment summary is issued after all other writes in a partial segment, the presence of the segment summary validates the partial segment. This approach requires two assumptions: the disk controller will not reorder the write requests and the disk writes the contents of a buffer in the order presented. Since controllers often reorder writes and reduce rotational latency by beginning track writes anywhere on the track, BSD-LFS can not make these assumptions. Therefore, segments are built from front to back, placing the segment summary at the beginning of each segment as shown in Figure 6-3. A checksum is computed across four bytes of each block in the partial segment, stored in the segment summary, and used to verify that a partial segment is valid. Figure 6-4 shows how this is done. This approach avoids write-ordering constraints and allows us to write multiple partial segments without an intervening seek or rotation. Currently, there is no data to indicate that this checksum is insufficient, however, methods exist for guaranteeing that any missing sector can be detected during roll-forward.

The most commonly used technique for verifying multi-sector disk writes uses *patch tables*. The patch table is a collection of bits that is stored for each multi-sector unit, or segment, in the case of LFS. Each time a segment is rewritten, the first bit in each sector is overwritten with either a 0 or 1 (alternating on each segment write). The real values for those bits are then stored both in the segment usage table and the segment summary block. During normal operation, when a block is read from disk, the real values for the first bit in each sector are read from the segment summary table and placed into the data blocks. During recovery, a segment is valid only if the

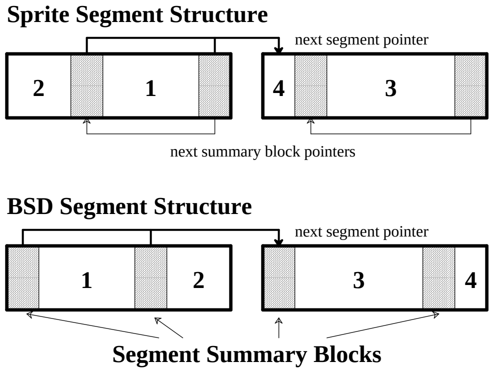


Figure 6-3: Partial Segment Structure Comparison Between Sprite-LFS and BSD-LFS. The numbers in each partial segment show the order in which the partial segments are created. Sprite-LFS builds segments back to front, chaining segment summaries. BSD-LFS builds segments front to back. After reading a segment summary block, the location of the next segment summary block can be easily computed.

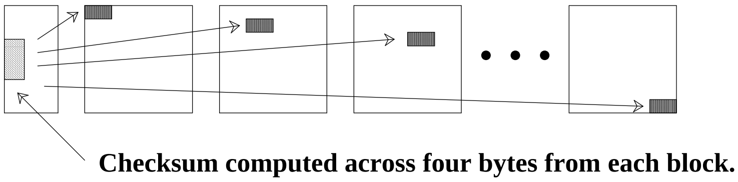


Figure 6-4: BSD-LFS Checksum Computation. A checksum is calculated on four bytes from every block in the segment to verify that the segment described by a summary block has been successfully written to disk. In order for the checksum to fail, the system must crash during a write, and the segment summary must have been successfully written, but for each unwritten data block, the old block on disk at the new block's location must have contained four bytes that checksum to the same value.

first bit on every sector is identical. Therefore, a segment's validity can be guaranteed at the expense of 2 bits per sector, or .04% overheard.

6.2.4. File System Verification

Fast recovery from system failure is desirable, but reliable recovery from media failure is necessary. Consequently, the BSD-LFS system provides two recovery strategies. The first quickly rolls forward from the last checkpoint, examining data written between the last checkpoint and the failure. The second does a complete consistency check of the file system to recover lost or corrupted data, due to the corruption of bits on the disk or errant software writing bad data to the disk. This check is similar to the functionality of *fsck,* the file system checker and recovery agent for FFS, and like *fsck*, it takes a long time to run.

In order for LFS to be viable in a production environment, it must make reliability guarantees comparable to FFS, which is an extremely robust file system. In the standard 4BSD implementation, it is possible to clear the root inode and recover the file system automatically with *fsck*(8). In terms of recovery, the advantage of LFS is that writes are localized, so the file system may be recovered to a physically consistent state very quickly.

The BSD-LFS implementation permits LFS to recover quickly and applications to start running as soon as the roll-forward has been completed, while basic sanity checking of the file system is done in the background. It may be more desirable to run the background file system checker periodically during normal operation, rather than waiting for a system crash or reboot. However, if the system crashed due a file system corruption, the verification will undoubtedly have to be run before any other processing can occur. Of course, the root file system must always be completely checked after every reboot, in case a system failure corrupted it.

There is the obvious problem of what to do if the sanity check fails. It is expected that the file system will forcibly be made read-only, fixed, and then write enabled. These events should have a limited effect on users as it is unlikely to ever occur and is even more unlikely to discover an error in a file currently being written by a user, since the opening of that file would most likely have already caused a process or system failure.

6.2.5. The Cleaner

In Sprite-LFS the cleaner is part of the kernel and implements a single cleaning policy. There are three problems with this, in addition to the memory issues discussed in Section 6.2.1. First, there is no reason to believe that a single cleaning algorithm will work well on all workloads. In fact, the transaction processing benchmark in Chapter 6 suggests that coalescing randomly updated files would improve sequential read performance. Second, placing the cleaner in kernel-space makes it difficult to experiment with alternate cleaning policies. Third, implementing the cleaner in the kernel forces the kernel to make policy decisions (the cleaning algorithm) rather than simply providing a mechanism. To handle theses problems, the BSD-LFS cleaner is implemented as a user process.

The BSD-LFS cleaner communicates with the kernel via system calls and the read-only *ifile*. Those functions that are already handled in the kernel (e.g. translating logical block numbers to disk addresses via *bmap*) are made accessible to the cleaner via system calls. If necessary functionality did not already exist in the kernel (e.g. reading and parsing segment summary blocks), it was relegated to user space.

There may be multiple cleaners, each implementing a different cleaning policy, running in parallel on a single file system. Regardless of the particular policy, the basic cleaning algorithm works as follows:

1. Choose one or more target segments and read them.
2. Decide which blocks are still ‘‘live’’.
3. Write ‘‘live’’ blocks back to the file system.
4. Mark the segment(s) clean.

The *ifile* and four new system calls, summarized in Table 6-4, provide the cleaner with enough information to implement this algorithm. The cleaner reads a regular file maintained by the kernel, called the *ifile*, to find out the status of segments in the file system. Using the information in the *ifile*, it selects segments to clean. Once a segment is selected, the cleaner reads the segment from the raw partition and uses the first segment summary to find out what blocks reside in that partial segment. It constructs an array of BLOCK_INFO structures (shown in Figure 6-5) and continues scanning partial segments, adding their blocks to the array. When all the segment summary block have been read, and all the BLOCK_INFOs constructed, the cleaner calls *lfs_bmapv* which returns the current physical disk address for each BLOCK_INFO. If the disk address is the same as the location of the block in the segment being examined by the cleaner, the block is

lfs_bmapv	Take an array of inode number/logical block number pairs and return the disk address for each block. Used to determine if blocks in a segment are ‘‘live’’.
lfs_markv	Take an array of inode number/logical block number pairs and append them into the log. This operation is a special purpose write call that rewrites the blocks and inodes without updating the inode’s access or modification times. The user process has already read the data from disk, so the kernel can copy the blocks from the user instead of re-reading them from disk.
lfs_segwait	Causes the cleaner to sleep until a given timeout has elapsed or until another segment is written. This operation is used to let the cleaner pause until there may be more segments available for cleaning.
lfs_segclean	Mark a segment clean. After the cleaner has rewritten all the ‘‘live’’ blocks from a segment, the segment is marked clean for reuse.

Table 6-4: The System Call Interface for the Cleaner.

‘‘live’’. Live blocks must to be written back into the file system without changing their access or modify times, so the cleaner issues an *lfs_markv* call, which is a special write causing these blocks to be appended into the log without updating their inode times.

Before rewriting the blocks, the kernel verifies that none of the blocks have ‘‘died’’ since the cleaner called *lfs_bmapv*. Once lfs_markv begins, only cleaned blocks are written into the log, until lfs_markv completes. Therefore, if cleaned blocks die after lfs_markv verifies that they are alive, partial segments written after the lfs_markv partial segments will reflect the fact that the blocks have died.

When *lfs_markv* returns, the cleaner calls *lfs_segclean* to mark the segment clean. Finally, when the cleaner has cleaned enough segments, it calls *lfs_segwait*, sleeping until the specified

INODE NUMBER
LOGICAL BLOCK NUMBER
CURRENT DISK ADDRESS
SEGMENT CREATION TIME
BUFFER POINTER

Figure 6-5: BLOCK_INFO Structure used by the Cleaner. The cleaner calculates the current disk address for each block from the disk address of the segment. The kernel specifies which have been superceded by more recent versions.

timeout elapses or a new segment is written into an LFS.

Since the cleaner is responsible for producing free space, the blocks it writes must get preference over other dirty blocks to be written to avoid running out of free space. However, it is possible for the cleaner to consume more disk space than it frees during cleaning. Although this cannot happen over the long-term, during the short-term it can. Consider the simple three segment file system shown below in Figure 6-6. Segment 1 contains one free block (the first block marked ‘‘Deleted Data’’). However, cleaning segment 1 requires rewriting the indirect block for file 1. Therefore, after segment 1 is cleaned, segment 3 will be full, segment 1 will be clean, and one block in segment 2 will be ‘‘dead’’ (Figure 6-7). While the total number of live blocks on the system has not increased, it has not decreased either, and the act of cleaning the segment has not created any additional space. It is possible to construct a case where cleaning a segment actually decreases the amount of available space (consider a segment that contains N blocks from N different files, each of which is accessed via an indirect block and the indirect block resides in a different segment). Therefore two segments are reserved for the cleaner. One guarantees that the cleaner can run, and the second ensures that small overflows can be accommodated until more space is reclaimed.

The cleaning simulation results in [ROSE91] show that selection of segments to clean is an important design parameter in minimizing cleaning overhead, and that the cost-benefit policy defined there does extremely well for the simulated workloads. Briefly, each segment is assigned a cleaning *cost* and *benefit*. The *cost* to clean a segment is equal to:

$$1 + utilization$$

where utilization is the fraction of ‘‘live’’ data in the segment. The *benefit* of cleaning a segment is equal to:

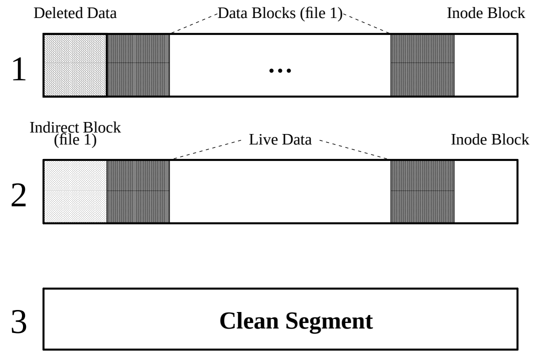


Figure 6-6: Segment Layout for Bad Cleaner Behavior. Segments 1 and 2 contain data. The cleaner will attempt to free up the one disk block of deleted data from segment 1. However, to rewrite the data in segment 1, it will dirty the meta-data block currently in segment 2. As a result, the cleaner will not generate any additional clean blocks.

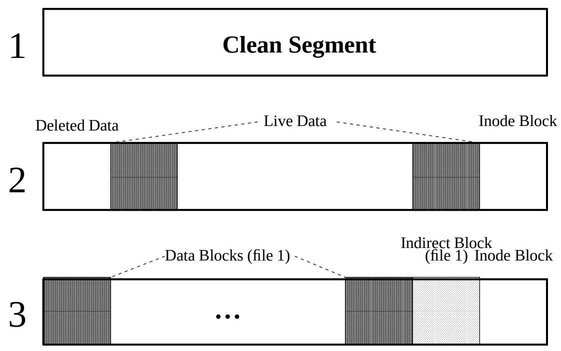


Figure 6-7: Segment Layout After Cleaning. The cleaner cleaned segment 1. In doing so, it rewrote the indirect block that previously resided in segment 2. Now that block has been deleted and the cleaner will be able to reclaim a disk block by cleaning segment 2.

$$\textit{free bytes generated} * \textit{age of segment}$$

where *free bytes generated* is the fraction of ‘‘dead’’ blocks in the segment (1 − *utilization*) and *age of segment* is the time since the most recent modification to a block in that segment. The age is incorporated into the benefit computation to avoid cleaning segments that are rapidly becoming empty. For example, consider that a segment contains a single, large file. If the file is being deleted, at some point before the entire file is deleted, the number of ‘‘live’’ blocks in that segment will fall below the minimum of all other segments. However, since the remaining blocks will soon be deleted, it is beneficial to wait for some period of time until the rest are deleted. Factoring age into the benefit computation serves this purpose. When the file system needs to reclaim space, the cleaner selects the segment with the largest benefit to cost ratio. BSD-LFS uses this as the default cleaning algorithm.

Currently the cost-benefit cleaner is the only cleaner running, but two additional policies are under consideration. The first would run during idle periods and select segments to clean based on coalescing and clustering files. The second would flush blocks in the cache to disk during normal processing even if they were not dirty, if it would improve the locality for a given file. These policies will be analyzed in future work.

6.3. Implementing LFS in a BSD System

While the last section focused on those design issues that addressed problems in the design of Sprite-LFS, this section presents additional design issues either inherent to LFS or resulting from the integration of an LFS into 4BSD.

6.3.1. Integration with FFS

The on-disk data structures used by BSD-LFS are nearly identical to the ones used by FFS. This decision was made for two reasons. The first one was that many applications have been written over the years to interpret and analyze raw FFS structures. It was desirable that these tools could continue to function as before, with minor modifications to read the structures from a new location. The second and more important reason was that it was easy and increased the

Operation	Description
mount	Mount a raw disk partition as a file system of the appropriate type.
start	Make a file system operational.
unmount	Remove a file system from the file system tree.
root	Return the root of a given file system.
quotactl	Perform quota operations.
statfs	Return file system statistics.
sync	Force all dirty in-memory buffers associated with this file system to disk.
fhtovp	Convert a file handle to a vnode.
vptofh	Convert a vnode to a file handle.
init	Initialize inode hash table for a file system.

Table 6-5: Description of Existing BSD *vfs* operations.

Operation	Description
lookup	Find the vnode for the specified named file.
create	Create a new file.
mknod	Make a new file node in the specified file system.
open	Open a file.
close	Close a file, freeing any of its allocated resources.
access	Check the permissions of the specified file.
getattr	Return a file's attributes.
setattr	Set a file's attributes.
read	Perform a read of the specified file.
write	Write to the specified file.
ioctl	Change the I/O characteristics of the specified file.
select	Check if the file is ready for reading or writing or have an exceptional condition pending.
mmap	Map the specified file into the virtual address space of the current process.
fsync	Force a file's blocks to disk.
seek	Move the file pointer for the specified file to the specified byte offset in the file.
remove	Delete one reference to a file (if it is the last reference, the file is deleted).
link	Create a link from a specified file to a new name.
rename	Change the name of a file.
mkdir	Create a directory.
rmdir	Remove a directory.
symlink	Create a symbolic link between the specified file and the specified name.
readdir	Return the next directory entry in the specified directory.
readlink	Return the destination of a symbolic link.
abortop	Abort a create or delete operation in progress.
inactive	Remove the last reference to a vnode.
reclaim	Free an inode in the inode hash table so it may be re-used.
lock	Lock a vnode.
unlock	Unlock a vnode.
strategy	Schedule I/O operations for the specified file.
print	Print out the contents of an inode.
islocked	Return true if the specified vnode is locked.
advlock	Lock/unlock advisory record locks.

Table 6-6: Description of existing BSD *vnode* operations.

maintainability of the system. A basic LFS implementation, without cleaner or reconstruction tools, but with *dumpfs*(1) and *newfs*(1) tools, was reading and writing from/to the buffer cache in under two weeks, and reading and writing from/to the disk in under a month. This implementation was done by copying the FFS source code and replacing about 40% of it with new code. The FFS and LFS implementations have since been merged to share common code.

In BSD and similar systems (e.g. SunOS, OSF/1), a file system is defined by two sets of interface functions, *vfs* operations and *vnode* operations [KLEI86]. *Vfs* operations affect entire file systems (e.g. mount, unmount, etc.) while *vnode* operations affect files (open, close, read, write, etc.). The original set of operations for each of these interfaces is described in Tables 6-5 and 6-6.

File systems could share code at the level of a vfs or vnode subroutine call, but they could not share the UNIX naming while implementing their own disk storage algorithms. To allow sharing of the UNIX naming, the code common to both FFS and BSD-LFS was extracted from the FFS code and put in a new, generic file system module (UFS). This code contains all the directory traversal operations, almost all vnode operations, the inode hash table manipulation, quotas, and locking. The common code is used not only by the FFS and BSD-LFS, but by the memory file system [MCKU90] as well. The FFS and BSD-LFS implementations remain responsible for disk allocation and actual I/O. Table 6-7 shows which of the *vnode* operations are file system specific. Any not included in Table 6-7 and those included only for LFS (the directory operations) are all in the common UFS code. LFS does some pre- and post- processing of directory operations and then calls the UFS routines.

In moving code from the FFS implementation into the generic UFS area, it was necessary to add seven new *vnode* and *vfs* operations. Table 6-8 lists the operations that were added to facilitate this integration and explains why they are different for the two file systems.

6.3.1.1. Block Sizes

One FFS feature that is not implemented in BSD-LFS is fragments. The original reason for fragments was that, given a large block size (necessary to obtain contiguous reads and writes, and to lower the data to meta-data ratio), fragments were required to minimize internal fragmentation (allocated space that does not contain useful data). LFS does not require large blocks to obtain contiguous reads and writes as it sorts blocks in a file by logical block number, writing them sequentially. Still, large blocks are desirable to keep the meta-data to data ratio low. Unfortunately, large blocks can lead to wasted space if many small files are present. Since managing fragments complicates the file system, BSD-LFS will allocate progressively larger blocks instead of using a block/fragment combination. This improvement has not yet been implemented but is similar to the restricted buddy policy simulated in Chapter 3.

6.3.1.2. The Buffer Cache

Prior to the integration of BSD-LFS into 4BSD, the buffer cache had been considered file system independent code. However, the buffer cache contains assumptions about how and when blocks are written to disk. First, it assumes that a single block can be flushed to disk, at any time, to reclaim its memory. There are two problems with this: flushing blocks a single block at a time would destroy any possible performance advantage of LFS, and, because of the modified meta-data and partial segment summary blocks, LFS may require additional memory to write. Therefore, BSD-LFS needs to guarantee that it can obtain any additional buffers it needs when it writes a segment.

To prevent the buffer cache from trying to flush a single BSD-LFS page, BSD-LFS does not put its buffers on the normal LRU queue, but puts them on the kernel LOCKED queue, so that the buffer cache cannot reclaim them. The number of buffers on the locked queue is compared against two variables, the *start write threshold* and *stop access threshold*, to prevent BSD-LFS from using up all the available buffers. When the number of LFS buffers on the LOCKED queue exceeds the *start write threshold*, the segment writer is invoked, and dirty buffers on the locked

File System	Operation
LFS	read, write, fsync, symlink, mknod, create, mkdir, remove, rmdir, link, rename
FFS	read, write, fsync

Table 6-7: Summary of File system Specific *vnode* **Operations.**

Vnode Operations	
blkatoff	Read the block at the given offset, from a file. The two file systems calculate block sizes and block offsets differently, because BSD-LFS does not implement fragments.
valloc	Allocate a new inode. FFS must consult and update bit maps to allocate inodes while BSD-LFS removes the inode from the head of the free inode list in the *ifile*.
vfree	Free an inode. FFS must update bit maps while BSD-LFS inserts the inode onto a free list.
truncate	Truncate a file from the given offset. FFS marks bit maps to show that blocks are no longer in use, while BSD-LFS updates the segment usage table.
update	Update the inode for the given file. FFS pushes individual inodes synchronously, while BSD-LFS accumulates them and writes them in a partial segment.
bwrite	Write a block into the buffer cache. FFS performs synchronous writes while BSD-LFS just marks the block dirty and puts it in the cache.
Vfs Operations	
vget	Get a vnode. FFS computes the disk address of the inode while BSD-LFS looks it up in the *ifile*.

Table 6-8: New Vnode and Vfs Operations. These routines allowed us to share 60% of the original FFS code with BSD-LFS.

queue will be written, marked no longer dirty, and removed from the locked queue. If the number of LFS buffers on the LOCKED queue exceeds the *stop access threshold*, then any requests, from LFS to obtain more buffers, are denied until the number of LFS buffers on the locked queue falls below the threshold. This problem can be much more reasonably handled by systems with better integration of the buffer cache and virtual memory.

Second, BSD maintains a logical block cache, hashed by vnode and logical block number. In FFS, since indirect blocks do not have logical block numbers, they are hashed by the vnode of the device (the file that represents the disk partition) and the disk block number. Since LFS does not assign disk addresses until blocks are written to disk, indirect blocks have no valid addresses on which to hash. To solve this problem, the block name space had to incorporate meta-data block numbering. This numbering is done by making block addresses be signed integers with negative numbers referencing indirect blocks, while zero and positive numbers reference data blocks. Figure 6-8 shows how the blocks are numbered. Singly indirect blocks take on the negative of the first data block to which they point. Doubly and triply indirect blocks take the next lower negative number of the singly or doubly indirect block to which they point. This approach makes it simple to traverse the indirect block chains in either direction, facilitating reading a block or creating indirect blocks. Sprite-LFS partitions the ‘‘block name space’’ in a similar fashion.

Although it is not possible for BSD-LFS to use FFS meta-data numbering, the reverse is not true. In 4.4BSD, FFS uses the BSD-LFS numbering and the *bmap* code has been moved into the UFS area.

6.3.2. The IFILE

Sprite-LFS maintained the inode map and segment usage table as kernel data structures which are written to disk at file system checkpoints. BSD-LFS places both of these data structures in a read-only regular file, visible in the file system, called the *ifile*. There are three advantages to this approach. First, while Sprite-LFS and FFS limit the number of inodes in a file system, BSD-LFS has no such limitation, growing the *ifile* via the standard file mechanisms. Second, it can be treated identically to other files, in most cases, minimizing the special case code in the operating system. Finally, it provides a convenient mechanism for communication between the operating system and the cleaner. A detailed view of the *ifile* is shown in Figure 6-9.

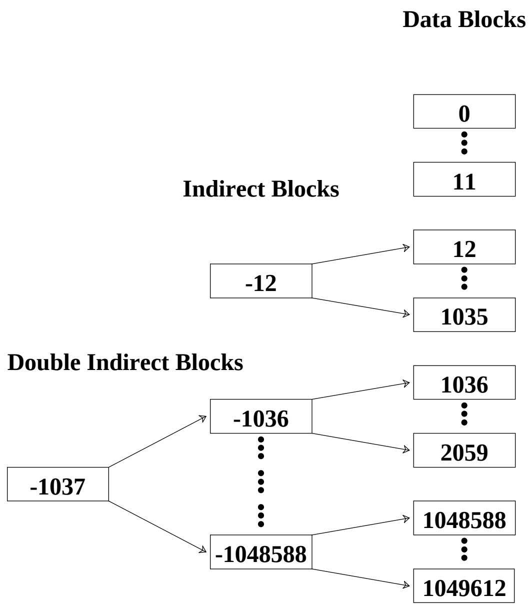


Figure 6-8: Block-numbering in BSD-LFS. In BSD-LFS, data blocks are assigned positive block numbers beginning with 0. Indirect blocks are numbered with the negative of the first data block that they address. Double and triple indirect blocks are numbered with one less than the first indirect or double indirect block that they address.

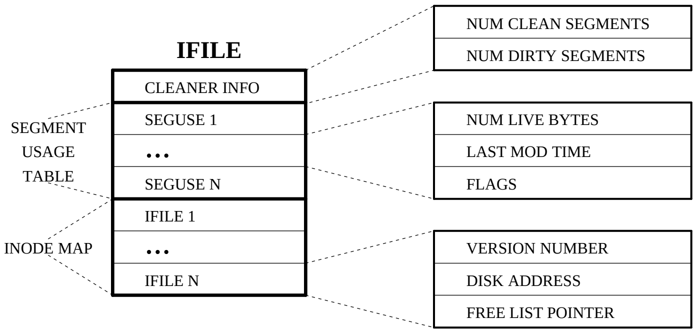


Figure 6-9: Detail Description of the IFILE. The *ifile* is maintained as a regular file with read-only permission. It facilitates communication between the file system and the cleaner.

Both Sprite-LFS and BSD-LFS maintain disk addresses and inode version numbers in the inode map. The version numbers allow the cleaner to easily identify groups of blocks belonging to files that have been truncated or deleted. Sprite-LFS also keeps the last access time in the inode map so that when files are read, the inode does not get rewritten and moved far away from the file data. However, since the access time is eight bytes in 4.4BSD, maintaining it in the inode map would cause the *ifile* to grow by 67%, so BSD-LFS keeps the access time in the inode.

Sprite-LFS clusters inodes in the inode map, and allocates new inodes by picking a starting point and scanning forward sequentially until it finds a free inode. To create a new file, the inode map is searched from the inode entry of the containing directory. If a directory is being created, a random location is chosen. When a directory contains many files this scan is costly. On six Sprite file systems, the average number of entries searched per directory or file creation ranged from 26 to 192, with an average across all the file systems of 94 entries per allocation. BSD-LFS avoids this scan by maintaining a free list of inodes in the inode map.

The segment usage table contains the number of live bytes in and the last modified time of each segment and is largely unchanged from Sprite-LFS. In order to support multiple and user mode cleaning processes, it also contains a set of flags indicating whether the segment is clean, contains a superblock, is currently being written to, or is eligible for cleaning.

6.3.3. Directory Operations

Directory operations[8] pose a special problem for LFS. Since the basic premise of LFS is that operations can be postponed and coalesced to provide large I/Os, it is counterproductive to retain the synchronous behavior of directory operations. At the same time, if a file is created, filled with data and *fsync*ed, then both the file's data and the directory entry for the file must be on disk. Additionally, the UNIX semantics of directory operations are defined to preserve ordering (i.e. if the creation of file *a* precedes the creation of file *b*, then any post-recovery state of a file system that includes file *b* must include file *a*). It is believed that this semantic is used in UNIX systems to provide mutual exclusion and other locking protocols.

[8] Directory operations include those system calls that affect more than one inode (typically a directory and a file) and include: create, link, mkdir, mknod, remove, rename, rmdir, and symlink.

Sprite-LFS preserves the ordering of directory operations by maintaining a directory operation log inside the file system log. Before any directory updates are written to disk, a log entry that describes the directory operation is written. The log information always appears in an earlier segment, or the same segment, as the actual directory updates. At recovery time, this log is read and any directory operations that were not fully completed are rolled forward. Since this approach requires an additional, on-disk data structure, and since LFS is itself a log, a different solution was chosen, namely *segment batching*.

Since directory operations affect multiple inodes (e.g. a new file and its containing directory), BSD-LFS must guarantee that either both of the inodes and associated changes get written to disk or neither does. BSD-LFS has a unit of atomicity, the partial segment, but it does not have a mechanism that guarantees that all inodes involved in the same directory operation will fit into a single partial segment. Therefore, a mechanism that allows operations to span partial segments is introduced. At recovery, a partial segment is never rolled forward if it has an unfinished directory operation and the partial segment that completes the directory operation did not make it to disk.

The requirements for segment batching are defined as follows:

1. If any directory operation has occurred since the last partial segment was written, the next segment write will append all dirty blocks from the *ifile* (that is, it will be a checkpoint, except that the superblock need not be updated).
2. During recovery, any writes that were part of a directory operation write will be ignored unless the entire write completed. A completed write can be identified if all dirty blocks of the *ifile* and its inode were successfully written to disk.

This definition is essentially a transaction where the writing of the *ifile* inode to disk is the commit operation. In this way, there is a coherent snapshot of the file system at some point after each directory operation. The penalty is that checkpoints are written more frequently in contrast to Sprite-LFS's approach that wrote additional logging information to disk.

The BSD-LFS implementation requires synchronizing directory operations and segment writing. Each time a directory operation is performed, the affected vnodes are marked and the memory-resident superblock is updated to reflect that a directory operation is in progress or has occurred. For example, the creation of file *f* in directory *d* can be decomposed into five steps:

1. File system state variable *dirop_in_progress* is set.
2. Vnode *d* is marked.
3. Vnode *f* is created and marked.
4. Vnode *d* is updated.
5. File system state variable *dirop* is set.
6. File system state variable *dirop_in_progress* is unset.

The segment writer uses the two state variables and the vnode markings to control its writing. When the segment writer builds a partial segment, it collects vnodes in two passes. In the first pass, all unmarked vnodes (those not participating in directory operations) are collected, and during the second pass those vnodes that are marked are collected. The segment writer will not begin pass two while the state variable *dirop_in_progress* is set, and directory operations are prohibited from beginning while the segment writer is in pass two. If any vnodes are found during the second pass, there are directory operations present in the current partial segment, and the segment summary block flags are set, identifying the partial segment as the beginning of a directory operation. The last partial segment containing marked vnodes is identified as completing the directory operation (in most cases, the beginning and ending identification will be in the same partial segment).

When recovery is run, the file system can be in one of three possible states with regard to directory operations:

1. The system shut down cleanly so that the file system may be mounted as is.
2. There are valid segments following the last checkpoint and the last one was a completed directory-operation write. Therefore, all that is required before mounting is to rewrite the superblock to reflect the address of the *ifile* inode and the current end of the log.
3. There are valid segments following the last checkpoint or directory operation write. As in the previous case, the system recovers to the last completed directory operation write and then rolls forward the segments from there to either the end of the log or the first partial segment beginning a directory operation that is never finished. Then the recovery process writes a checkpoint and updates the superblock.

While rolling forward, two flags are used in the segment summaries: SS_DIROP and SS_CONT. SS_DIROP specifies that a directory operation appears in the partial segment. SS_CONT specifies that the directory operation spans multiple partial segments. If the recovery agent finds a partial segment with both SS_DIROP and SS_CONT set, it ignores all such partial segments until it finds a later partial segment with SS_DIROP set and SS_CONT unset (i.e. the end of the directory operation write). If no such partial segment is ever found, then all the partial segments from the initial directory operation on are discarded. Since partial segments are small [BAKER92] this should rarely, if ever, happen.

6.3.4. Synchronization

To maintain the delicate balance between buffer management, free space accounting and the cleaner, synchronization between the components of the system must be carefully managed. Figure 6-10 depicts the synchronization relationships.

The cleaner is given precedence over all other processing in the system to guarantee that clean segments are available if the file system has space. It has its own event variable on which it waits for new work (*lfs_allclean_wakeup*). The segment writer and user processes will defer to the cleaner if the disk system does not have enough clean space. A user process detects this condition when it attempts to write a block but the block accounting indicates that there is no space available. The segment writer detects this condition when it attempts to begin writing to a new segment and the number of clean segments has reached two.

In addition to cleaner synchronization, the segment writer and user processes synchronize on the the availability of buffer headers. When the number of LFS buffer header on the LOCKED queue exceeds the *start write threshold* a segment write is initiated. If a write request would make the number of LFS buffers on the LOCKED queue exceed the *stop access threshold*, the writing process waits until a segment write completes, making more buffer headers available. Finally, there is directory operation synchronization. User processes wait on the *lfs_dirop* condition and the segment writer waits on *lfs_writer* condition.

6.3.5. Minor Modifications

There are a few additional changes to Sprite-LFS. To provide more robust recovery the superblock is replicated up to ten times throughout the file system, as in FFS. Since the file system meta-data is stored in the *ifile*, there is no need for separate checkpoint regions, and the disk address of the *ifile* inode is stored in the superblock. Note that it is not necessary to keep a duplicate *ifile* since it can be reconstructed from segment summary information, if necessary.

6.4. Conclusions

The implementation of BSD-LFS highlighted some subtleties in the overall LFS strategy. While allocation in LFS is simpler than in extent-based file systems or file systems like FFS, the management of memory is much more complicated. The Sprite implementation addressed this problem by reserving large amounts of memory. Since this was not feasible in our environment, a more complex mechanism to manage buffer and memory requirements was needed.

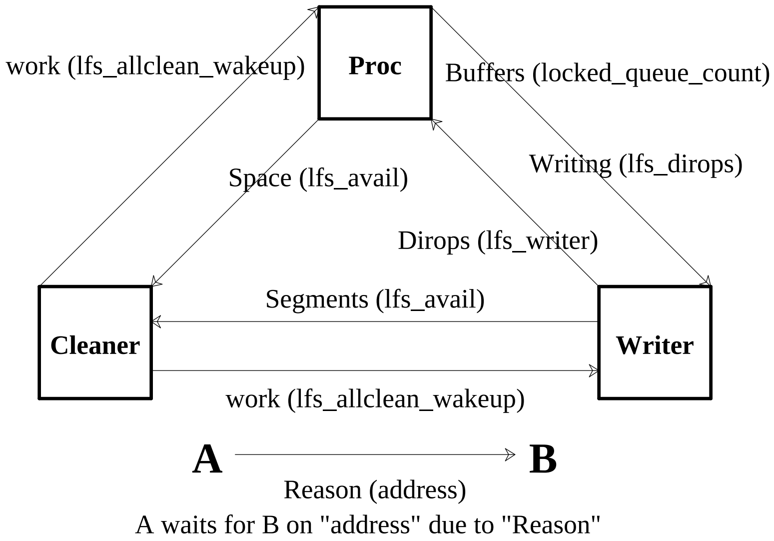


Figure 6-10: Synchronization Relationships in BSD-LFS. The cleaner has precedence over all components in the system. It waits on the *lfs_allclean_wakeup* condition and wakes the segment writer or user processes using the *lfs_avail* condition. The segment writer and user processes maintain directory operation synchronization through the *lfs_dirop* and *lfs_writer* conditions. User processes doing writes wait on the *locked_queue_count* when the number of dirty buffers held by BSD-LFS exceeds a system limit.

LFS operates best when it can write out large numbers of dirty buffers at once. However, holding dirty data in memory until a large amount has accumulated requires consuming more memory than might be desirable. In addition, the act of writing a segment requires allocation of additional memory (for segment summaries and on-disk inodes), so segment writing needs to be initiated before memory becomes a critical resource, in order to avoid memory thrashing.

The delayed allocation of LFS makes accounting of available free space more complex than in a pre-allocated system like FFS. In Sprite-LFS, the space available to a file system is the sum of the disk space and the buffer pool. As a result, files are allocated in the buffer pool for which there might not be free space available on disk. Since the applications that write these files may have exited before the files actually go to disk, there is no effective way to report the ‘‘out of disk space’’ condition. In order to avoid this phenomenon, available space accounting must be performed as dirty blocks enter the cache instead of when they are written from cache to disk.

This chapter has discussed the new design of LFS. The prior studies, both simulated and empirical have guided this redesign, and the re-implementation highlighted some difficult issues in building log-structured file systems. This new design attempts to avoid the disruption in service due to the cleaner and provides a mechanism by which the sequential file performance after random updates described in Chapter 5 can be improved. In addition, the new design has improved robustness and flexibility. The next chapter presents the performance evaluation of this new system.

Chapter 7

Performance Evaluation

This chapter compares the performance of the redesigned log-structured file system to more traditional, read-optimized file systems on a variety of benchmarks that attempt to emulate real workloads. The new log-structured file system was written in November of 1991, left largely untouched until late spring 1992, and is a completely untuned implementation.

The file systems against which LFS is compared are the regular fast file system (FFS), and an enhanced version of FFS introduced in [MCVO91] and described in the next section. The enhanced FFS file system is referred to as EFS for the remainder of this thesis.

7.1. Extent-like Performance Using the Fast File System

Chapter 3 showed that read-optimized policies that favor contiguous layout perform similarly, so in this chapter a variant of FFS that favors contiguous allocation was selected. The fundamental idea proposed in [MCVO91] is that the FFS block allocator can be used to allocate blocks contiguously on disk and that doing so allows the file system to read a large number of blocks, called a cluster, in a single I/O. In this way, EFS provides extent-based file system behavior without changing the underlying structures of FFS.

FFS is parameterized by a variable, *maxcontig*, which indicates how many logically sequential disk blocks should be allocated contiguously on disk. By setting *maxcontig* large (equal to a track or more), the FFS can be made to perform what is essentially track allocation. Logically sequential dirty buffers are accumulated in the cache, and when an extent's worth (i.e. *maxcontig* blocks) have been collected, they are bundled together into a cluster. This provides extent-based writing.

In order to provide extent-based reading, the interaction between the buffer cache and the disk was modified. Typically, before a block is read from disk, the *bmap* routine is called to translate logical block addresses to physical disk block addresses. The block is then read from disk and the next block is requested. Since I/O interrupts are not handled instantaneously, the disk is usually unable to respond to two contiguous requests on the same rotation, so sequentially allocated blocks incur the cost of an entire rotation. In EFS, *bmap* is extended to return not only the physical disk address but the number of contiguous blocks that follow the requested block. Then, rather than reading one block at a time and requesting the next block asynchronously, the file system reads a large number of the contiguous blocks in a single request. This provides extent-based reading.

This same mechanism is used by LFS to read its contiguously allocated disk blocks. However, because LFS potentially allocates more blocks contiguously than it can access in a single transfer (e.g. more than the maximum supported by the controller, typically 64 kilobytes), it may miss a rotation between reading collections of blocks. Disks with track buffers can often hide this rotational delay. Since EFS uses the FFS block allocator, it automatically leaves a rotational delay between clusters of blocks, and does not miss the rotation even in the cases where no track buffer is present or track-buffering fails.

7.2. The Test Environment

The hardware configuration consists of a Hewlett-Packard series 9000/380 computer with a 25 Mhz MC68040 processor. It has 16 megabytes of main memory, and a 1.3 gigabyte SCSI SD97560 disk. The hardware configuration is summarized in Table 7-1.

The system is running the 4.4BSD-Alpha operating system. The three file systems being evaluated all use four-kilobyte blocks, with EFS and FFS using one-kilobyte fragments. They all run in the same operating system kernel and share most of their source code. There are approximately 6000 lines of shared C code, 4000 lines of LFS-specific code, and 3500 lines of FFS-specific code. EFS uses the same source code as FFS plus an additional 500 lines of clustering code, of which 300 are also used by LFS (for reads). All measurements were taken with the system running single-user, with no network connections.

Five types of statistics were collected to analyze performance. First, there is a metric for each benchmark that quantifies the performance of each file system. The metrics are either elapsed time or throughput, measured in megabytes per second, files per second, or transactions per second.

Next there are counters in the kernel to measure disk activity. All the tests are run on the SCSI disk described in Table 7-1, and that disk is unused except for the benchmark, so the disk activity summarizes all the I/O performed by the benchmarks. The kernel maintains the number of reads and writes, the number of synchronous reads and writes, the total number of sectors read and written, and the microseconds that the disk is busy and idle. It also maintains a list of the last 1000 I/O begin and completion times from which individual I/O response times can be computed.

Disk (SCSI SD97560)		Comment
Average seek	13.0 ms	
Single rotation	15.0 ms	
Track size	36 KB	
Track buffer	128 KB	non-volatile; read-buffering only
Disk bandwidth	2.2 MB/sec	
Bus bandwidth	1.6 MB/sec	
Controller overhead	1.0 ms	
Track skew	8 sectors	
Cylinder skew	10 sectors	total skew = track skew + cylinder skew
Cylinder size	19 tracks	684 KB
Disk size	1962 cylinders	1.3 gigabytes
CPU (Motorola 68040)		
Memory Bandwidth	12.0 MB/sec	with 5000 byte transfers
CPU	25 Mhz	
MIPS	10-12	

Table 7-1: Hardware Specifications. Although the disk can support 2.2 megabytes per second transfer bandwidth, the SCSI interface is limited to 1.6 megabytes per second. SCSI supports two transfer modes, synchronous and asynchronous [ADAP85]. Synchronous mode is optional under SCSI-I and is not supported by the disk driver. Therefore, all transfers are performed using asynchronous mode and are limited to 1.6 MB/sec.

The kernel also maintains counters to monitor LFS activity. It counts the number of segments used, the number of checkpoints, the number of writes issued, the number of partial segments written, the number of partial segments from the cleaner, and what percentage of writes are synchronous. It also keeps track of the total number of blocks written and how many of those were due to the cleaner. Finally it keeps track of how many times the *start write threshold* and *stop access threshold* are reached.

Since the LFS cleaner runs as a user-process, it maintains its own set of counters, specifically the number of blocks read during cleaning, the number of block writes passed to the kernel, the number of segments that were empty and could be reclaimed without cleaning, and the number of segments cleaned.

The kernel also maintains a count of the number of blocks written by the cleaner, and its count can be used to verify the cleaner's count. The cleaner may submit blocks to the kernel for writing that have become invalid and therefore will not be written, so the cleaner's number of blocks written should be greater than or equal to the kernel's number of blocks written.

Finally, there are measurements derived from running the benchmark on a *profiling* kernel. Kernel profiling requires that a special monitoring routine is called on entry and exit to every subroutine. The kernel keeps track of the number of times each routine is called and these statistics can be reset and displayed upon request. As the act of profiling can be disruptive to system behavior (it incurs approximately a 12% overhead on the CPU), all other measurements, including the benchmark metrics, are reported for a non-profiling kernel. Profiling was used only when none of the other measurements could explain the system behavior.

Each of the next sections describes a benchmark and presents the performance analysis for each file system. With the exception of the first two benchmarks (raw file system performance and small file performance), the benchmarks attempt to model specific workloads described in Table 7-2.

7.3. Raw File System Performance

The goal of this test is to measure the maximum throughput that can be expected from the given disk and system configuration for each of the file systems. For this test, the three file systems are compared against the maximum speed at which the operating system can write directly to the disk. The benchmark consists of creating a file of size *S* and then either reading or writing

Benchmark	Section	Workload/Purpose
Raw	7.3	Measure the maximum throughput that the file system can achieve.
Small File	7.4	Measure the LFS benefits in processing a large number of small files.
Andrew	7.5	Software development workload.
OO1	7.6	Object oriented database workload.
Wisconsin	7.7	Complex query processing workload.
TPC/B	7.8	Transaction processing workload.
Super	7.9	Super computer workload.

Table 7-2: Summary of Benchmarks Analyzed.

the entire file 50 times. The measurements recorded are averages across the 50 runs. For the read tests, the cache is flushed before each test by unmounting and remounting the file system.

7.3.1. Raw Write Performance

The graph in Figure 7-1 shows the bandwidth attained for writing, as a function of *S*, the size of the I/O. Given the sequential layout of both LFS and EFS, the expectation is that both should perform comparably to the speed of the raw disk and that FFS, with its rotational positioning, should achieve approximately 50% of the disk bandwidth. However, there are several anomalies.

First, as the I/O size increases, EFS actually provides more bandwidth than the raw disk partition. The explanation for this can be found by looking at the number of synchronous I/O's and the begin time for each operation. When accessing the raw partition, all I/Os are synchronous. Therefore, there is no overlap between the time required to copy the data from user-space into the kernel and the time required to perform the I/Os. As a result, there is a gap of approximately five milliseconds between the completion of each I/O and the initiation of the next I/O. In contrast, EFS has an aggressive buffering policy, allowing it to perform asynchronous writes in units of 64 kilobytes. Therefore, the I/Os are queued, and successive I/Os are begun almost immediately.

The next anomaly lies in the fact that LFS performs noticeably worse than either EFS or the RAW partition. This is an artifact of this benchmark as opposed to a fundamental difference in the attainable write bandwidth of the two file systems. The problem is that the benchmark performs the write and then calls *fsync* to ensure that the blocks have been written to disk.

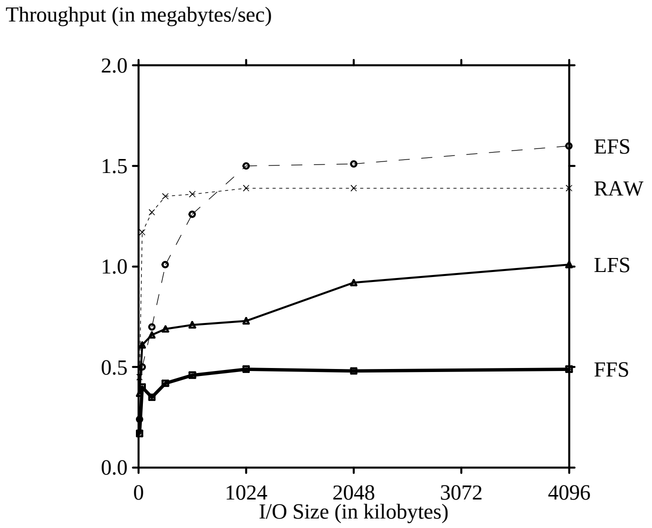


Figure 7-1: Maximum File System Write Bandwidth. This graph shows the write bandwidth of each file system as a function of the transfer size. EFS attains the best performance, as it performs nearly all its writes asynchronously in maximal-sized buffers. Writes to the RAW partition also occur in maximal-sized units, but are performed synchronously. In LFS, since a large amount of data are gathered in the cache before being written to the disk, there is less overlap between CPU processing and disk activity, leading to the gap shown above. The rotational delay of FFS prohibits it from achieving more than 25% of the available disk bandwidth.

LFS achieves its write performance by buffering a large number of dirty buffers before initiating I/O. As a result, LFS does not begin writing any data to disk until requested to do so by the application *fsync* or until the *start write threshold* has been reached. On this system, the *start write threshold* results in approximately 800 kilobytes of data being buffered before a write is initiated. As a result, for all the tests where the transfer size was smaller than one megabyte, the benchmark had two phases, the first in which data was written into the cache, and the second during which time the data was being written to disk.

To verify this, timings were taken after all the writes had been issued, but before the call to *fsync* and then again after the call to *fsync*. In the tests where the total transfer size was less than 800K, LFS' elapsed time for the *fsync* was nearly identical to the time required for EFS to write all its buffers. Figure 7-2 depicts this behavior. In the tests where the transfer size was greater than 800K, LFS' elapsed time for the *fsync* was the time reported for the synchronous LFS write that flushed the data remaining in the cache at the time of the *fsync*.

These write tests were repeated for LFS with the cleaner running, but the results were indistinguishable from the results without the cleaner. Since the same data is overwritten for each iteration of the test, there are always empty segments available for reclamation by the cleaner. As a result, the cleaner reported that it always cleaned empty segments, and the overhead was unmeasurable.

The last anomaly is that FFS did not achieve the 50% bandwidth expected, but achieved closer to 31% of the transfer bandwidth (0.5 megabytes per second of the possible 1.6 megabytes per second). The explanation of this is in the FFS *rot_delay* parameter. This parameter is used by FFS to express the length of time, from the disk's perspective, that it takes the CPU to acknowledge the completion of and I/O and to issue another one.[9]

For the system under test, the *rot_delay* that provided the best performance was experimentally determined to be 4 milliseconds. This value was determined by building file systems with successively larger *rot_delays* and selecting the value that led to the best performance. However, with a rotational latency of 15 milliseconds, 36 kilobyte tracks, 4-kilobyte blocks, and a 4 millisecond *rot_delay*, only one in four blocks is allocated to the same file, as shown in Figure 7-3. The maximum transfer bandwidth of the disk is 2.2 megabytes per second and one-quarter of this

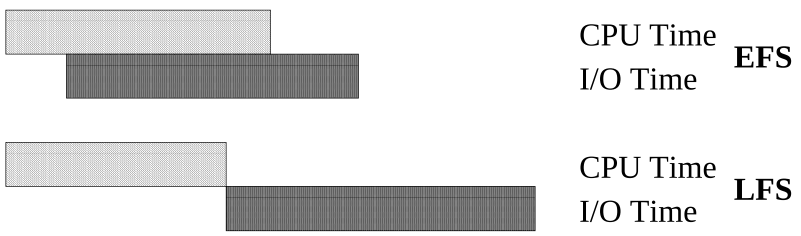


Figure 7-2: Effects of LFS Write Accumulation. The bars represent elapsed time for each phase of the benchmark on a one-half megabyte write. EFS effectively overlaps I/O and CPU processing while LFS waits until all the data is accumulated before initiating the write. As a result, the bandwidth measured by this test appears much lower for LFS.

[9] This is based on the assumption that queueing is performed by the host and not the disk.

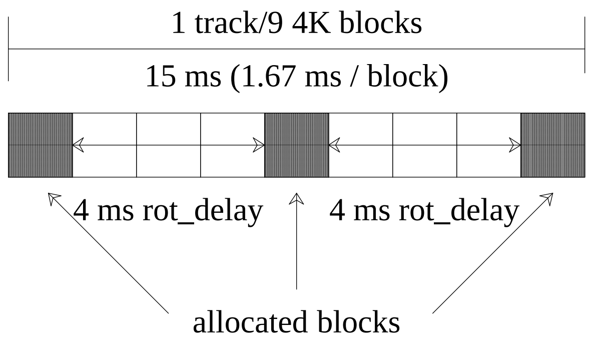


Figure 7-3: Impact of Rotational Delay on FFS Performance. Since *rot_delay* for this disk is 4 milliseconds, FFS will allocate only one in every four blocks. Therefore, at most 3 blocks (2.25 on average) can be accessed on each disk rotation. Therefore, FFS will attain at most one-quarter of the maximum bandwidth of the disk.

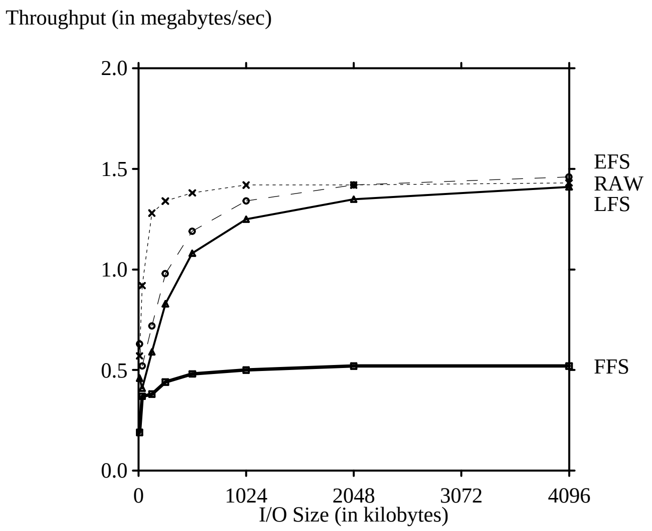


Figure 7-4: Maximum File System Read Bandwidth. The graph shows the maximum read throughput attained by each file system as a function of the transfer size. As EFS and LFS allocate blocks contiguously and use the exact same read-ahead algorithm, the expectation is that both will perform comparably to the raw partition. Once again, FFS is limited to approximately 25% of the total disk bandwidth due to rotational delays between allocated blocks.

is 0.55 megabytes per second, which is close to the observed performance of FFS.

7.3.2. Raw Read Performance

The results of the raw read tests, shown in Figure 7-4, are much closer to what is expected. FFS demonstrates read performance nearly identical to its write performance since it is limited by the number of blocks transferred during a single rotation. Both LFS and EFS perform comparably to the raw disk with very small (3%) differences in performance.

This benchmark demonstrates that both EFS and LFS can utilize close to 100% of the available I/O bandwidth on large I/Os. When individual write response time is an issue, LFS incurs a performance penalty due to its delayed write policy.

7.4. Small File Performance

This test evaluates the performance of LFS for small file processing. The benchmark consists of creating 10,000 files, of one kilobyte each, reading the 10,000 files, and then deleting them. In order to avoid spending the entire benchmark performing directory lookup, the files are created in 100 different directories, each containing 100 files. After the creation phase, the file system is unmounted to ensure that all the files have been written to disk. The read test begins with an empty cache (guaranteed by mounting the unmounted file system) and reads the files in the order in which they were created. The delete test also begins with an empty cache and deletes each file, unmounting the file system to ensure that all the required updates are on disk. Each test is run 10 times (10,000 files on each test) and the results shown in Figure 7-5 are the averages across the 10 runs. The standard deviations were within 0.1% to 0.2% of the elapsed time.

As expected, the asynchronous file creation and deletion of LFS makes it the clear winner in the create and delete phases. However, its poorer performance in the read test is surprising given that the files are created and read in the same order. The difference in read performance is a result of the fact that BSD-LFS does not currently support fragments. With four-kilobyte blocks, LFS transfers four times the amount of data that EFS or FFS transfer and its performance is approximately half that of EFS and FFS. In order to compensate for this, LFS used a block size of one kilobyte for these tests. However, LFS uses the block size as the unit of allocation for

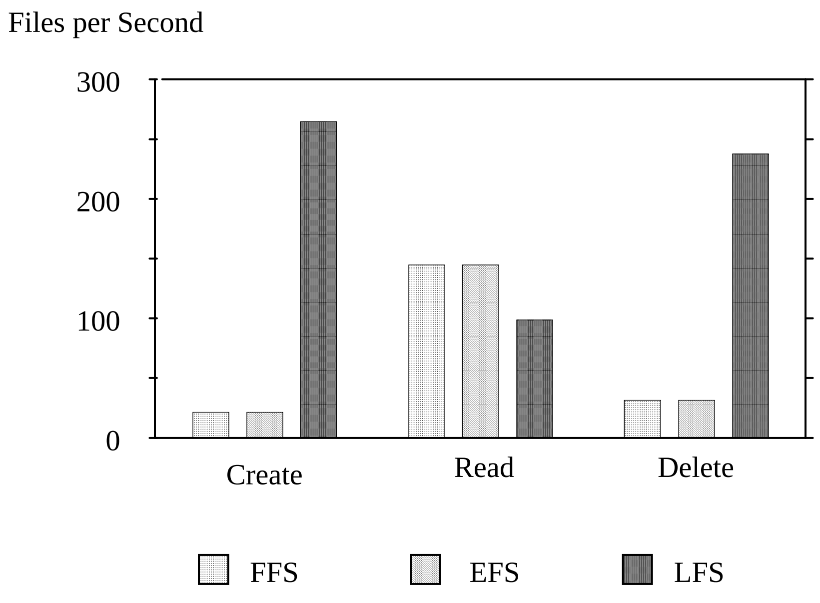


Figure 7-5: Small File Performance. This graph uses the metric ‘‘files per second’’ to depict the create, read, and delete times for the three file systems. As expected, the asynchronous creation and deletion of LFS make it the obvious winner in these tests. However, the lack of fragments in LFS makes the read performance approximately 30% worse than the other file systems.

inodes as well. Therefore, while EFS and FFS read 32 inodes per I/O (inodes are 128 bytes), LFS reads only 8 inodes per I/O and performs four times the number of reads on inode blocks. This number accounts for the difference in the number of reads reported for the two tests and explains the reduced read performance.

This test shows that LFS excels on small file performance and that the lack of fragments incurs not only a penalty in terms of fragmentation, but also in terms of performance when files are very small.

7.5. Software Development Workload

The next tests evaluate the file systems in a typical software development environment. The Andrew benchmark [HOWA88] is often used for this type of measurement. It contains five phases.

1. Create a directory hierarchy.
2. Make one copy of the data.
3. Recursively examine the status of every file in the test set.
4. Examine every byte of every file in the test set.
5. Compile several of the files in the test set.

Unfortunately, the test set for the Andrew benchmark is small, and main-memory file caching can make the results uninteresting. In order to exercise the file systems, this benchmark is run both single-user and multi-user (where several invocations of the benchmark are run concurrently).

7.5.1. Single-User Andrew Performance

Table 7-3 shows the performance of the standard Andrew benchmark. The entire five-phase test was run ten times for each of FFS, EFS, and LFS, with the directory hierarchy deleted after each pass. For the LFS test with the cleaner running (LFSC), the test was repeated 100 times to ensure that the file system was completely overwritten at least twice. In order to understand the differences in performance, the kernel counters for the disk and for LFS were initialized before, and sampled after, each phase.

	Phase 1 Create Directories	Phase 2 Copy Files	Phase 3 Stat Touch Inodes	Phase 4 Grep Touch Bytes	Phase 5 Compile	Total
FFS	2.10 (0.30)	7.90 (0.30)	6.30 (0.46)	9.00 (0.00)	44.80 (0.40)	70.1
EFS	2.10 (0.30)	7.90 (0.30)	6.70 (1.19)	9.10 (0.30)	44.40 (0.49)	70.2
LFS	0.33 (0.47)	5.00 (0.00)	6.50 (0.81)	9.07 (0.25)	42.90 (1.40)	63.8
LFSC	0.43 (0.49)	5.09 (0.28)	6.37 (0.48)	9.07 (0.26)	42.61 (0.49)	63.6

Table 7-3: Single-User Andrew Benchmark Results. This table shows the elapsed time for each phase of the benchmark on each file system. Reported times are the average across ten iterations with the standard deviation in parentheses. LFSC indicates that the benchmark was run on the log-structured file system with the cleaner running, but the similarity in results for most phases indicates that the cleaner had virtually no impact on performance. Overall, LFS demonstrates approximately a 9% difference in performance which can be attributed to asynchronous file creation and write-clustering.

Overall, LFS demonstrates a 9% improvement over EFS and FFS, which perform comparably. The difference is isolated to phases one, two, and five. It is not surprising that LFS would outperform the other systems in phase one, the create phase, as LFS performs all its directory creations asynchronously, performing no writes, while EFS and FFS issue 100 synchronous writes each. As phase two is the write-intensive phase, it is also expected that LFS will perform better, and it does so, demonstrating 37% better performance than the other two systems. Again, EFS and FFS are performing a great deal of I/O (263 requests for about 750 kilobytes), over half of which are synchronous (as a result of closing files). LFS performs no writes during this phase as all the data is written to the cache.

Phase five, which is moderately CPU-intensive (utilizations for LFS are approximately 59% and utilizations for EFS are approximately 49%), surprisingly demonstrates a small (3-5%) advantage for LFS. Once again, the disk kernel counters reveal that EFS and FFS are synchronously writing the output object files to disk (45 of 48 writes) while LFS is buffering the data and performing nearly one-third the number of writes.

The more striking difference is in the number of reads issued by the two file systems in phase five. LFS issues only a single read while FFS issues 46 of them. The explanation for this also lies in file allocation. When FFS creates a file, it allocates an inode from the appropriate cylinder group and then reads the contents of the inode from disk.[10] In LFS, since inodes do not reside in any permanent location on disk, new inodes are created in memory, not read from the disk.

These single-user results differ slightly from those presented in [ROSE92]. First, the compilation phase in [ROSE92] is much longer than in this test because different compilers were used. Secondly, the results in [ROSE92] show LFS providing a 40% performance improvement on phase three (the phase that examines every inode) and a 29% performance improvement on phase four (the phase that examines every byte), while the results here show virtually no difference. Phases three and four perform no I/O on any of the file systems, so performance is limited strictly by the file system code that reads data from the cache, traverses directories, and reads inodes from the in-memory inode cache. Since the three file systems share the same code for performing these functions, the expectation is that the systems should behave identically. Since the system measured in [ROSE92] is unavailable for instrumentation, it is unclear why results on phases three and four differ.

7.5.2. Multi-User Andrew Performance

The multi-user version of Andrew shows the file system performance as a function of the degree of multiprogramming. The test is performed by running N concurrent invocations of the benchmark, with each invocation creating, traversing, and removing its own directory hierarchy. The reported results are the averages of the results of each of ten runs for each invocation. The resulting averages are divided by the multiprogramming level to produce the metric ‘‘elapsed seconds per invocation.’’

The goal of the multi-user test is to examine two aspects of the file systems’ behavior under the software development workload. First, as the multiprogramming level increases, the entire data set no longer fits in the cache, so more I/O is performed. Secondly, with separate directory hierarchies, the different forms of locality used by LFS (temporal locality -- files created at about the same time reside close together) and FFS (logical locality -- files in the same directory are placed close together) can be compared.

The multi-user performance is the result of two competing factors. As concurrent invocations of the benchmark compete for resources, the utilization of both the CPU and the disk increases, as does performance. However, after the multiprogramming level exceeds two, the total working set becomes too large to fit in the cache and the total I/O time increases. Towards the left-hand side

---

This is an artifact of the file system architecture and could be avoided by modifying the interface to the *vfs* routine *vfs_vget*.

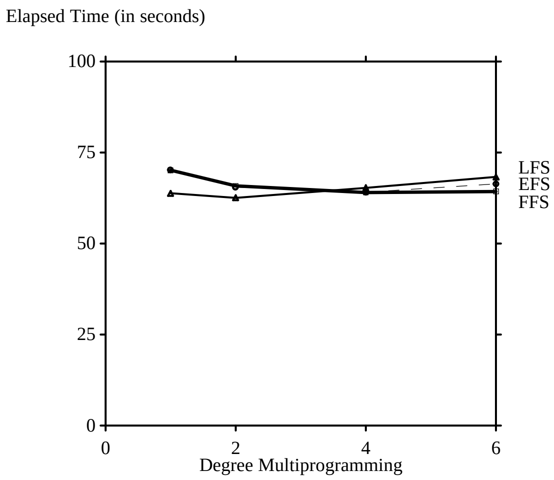


Figure 7-6: Multi-User Andrew Performance. This graph shows the elapsed time for all five phases of the Andrew benchmark under increasing multiprogramming. Overall, the impact of multiprogramming is less significant than might have been expected, yielding at most a 9% performance improvement.

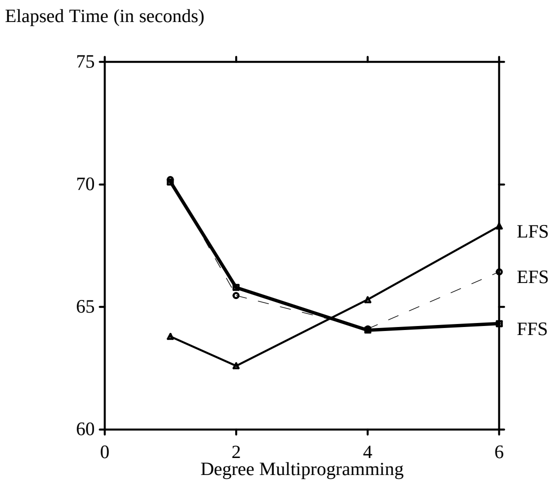


Figure 7-7: Multi-User Andrew Performance (Blow-Up). This graph emphasizes the small performance differences in the multi-user Andrew benchmark. There are two effects at work in this test. For EFS and FFS which perform many synchronous operations, multi-programming allows the overlapping of CPU and disk (evidenced by higher utilization for both resources) and reduced per-invocation time. LFS also benefits from this overlap, but not as significantly as the other systems. The second effect is that the total data set size begins to exceed the cache capacity and read performance becomes more important.

of the graph, the predominant factor is the overlap between CPU and disk. As LFS is already performing most of its I/O asynchronously, it has less room for improvement than EFS and FFS. So, the CPU utilization for LFS increases from 60% to 80% while the CPU utilization for EFS goes from 50% to 89%, explaining the steeper decline in elapsed time for EFS than for LFS.

As the multiprogramming level exceeds four, the data sets no longer fit in the cache and for all the systems the read performance becomes the dominant factor. Kernel I/O statistics reveal that on average, LFS is performing more seeks than EFS, explaining the small difference in performance observed as the multiprogramming level increases.

This benchmark indicates that LFS and EFS perform comparably on this particular software development workload. To generalize, LFS demonstrates superior file creation performance, but logical locality appears better than temporal locality when the working set is too large to fit in the cache.

7.6. OO1 -- The Object Oriented Benchmark

The next set of tests come from database environments. The first is the object oriented database benchmark, OO1, described in [CATT91]. The database models a set of electronics components with connections among them. One table stores parts and another stores connections. There are three connections originating at any given part. Ninety percent of these connections are to nearby parts (those with nearby *ids*) to model the spatial locality often exhibited in CAD applications. Ten percent of the connections are randomly distributed among all other parts in the database. Every part appears exactly three times in the *from* field of a connection record, and zero or more times in the *to* field. Parts have *x* and *y* locations set randomly in an appropriate range.

The intent of OO1 is to measure the overall cost of a query mix characteristic of engineering database applications. There are three tests:

- *Lookup* generates 1,000 random part *ids*, fetches the corresponding parts from the database, and calls a null procedure in the host programming language with the parts' *x* and *y* positions.
- *Traverse* retrieves a random part from the database and follows connections from it to other parts. Each of those parts is retrieved, and all connections from it followed. This procedure is repeated depth-first for seven hops from the original part, for a total of 3280 parts. Backward traversal also exists, and follows all connections to a given part back to their origins.
- *Insert* adds 100 new parts and their connections.

The benchmark is single-user, but multi-user access controls (locking and transaction protection) must be enforced. It is designed to be run on a database with either 20,000 parts or one with 200,000 parts. These measurements here are for the 20,000 part database which results in the database of 14.5 megabytes detailed in Table 7-4. The transaction package described in [SELT92] was used to create B-tree and fixed-length record files.

The test specification calls for running from both cold and warm caches, however, since the performance of the file and disk system is of interest, only the cold cache tests are run. While it is possible to ensure that there are no pages in the database cache by stopping and restarting the application, making the same guarantees for the operating system cache is accomplished by recopying the database from scratch and reading several unrelated files through the cache. Table

Relation	Format	Entry Size	N Entries	Total Size
Part	B-tree records	63	20000	2.5 MB
Connect Forward	B-Tree	43	60000	3.7 MB
Connect Backward	B-Tree	43	60000	2.5 MB
Connector	Fixed-length Record	96	60000	5.8 MB
Total				14.5 MB

Table 7-4: Database Sizing for the OO1 Benchmark. This table shows how the database sizes reflect the number of entries in each relation. The column ‘‘entry size’’ reflects the number of valid data bytes plus the entry overhead (28 bytes for each B-tree entry, 4 bytes for each fixed-length record). B-Trees that are created in-order (Connect Forward and Part) result in all but one page being half-empty, because when pages split, the page containing the smaller keys never has any more keys appended to it.

	Lookup	Insert	Traverse	
			Forward	Backward
FFS	16.30 (0.46)	4.40 (0.49)	28.10 (0.54)	28.60 (0.49)
EFS	16.40 (0.49)	4.20 (0.40)	28.40 (0.49)	28.80 (0.60)
LFS	16.30 (0.64)	4.30 (0.46)	28.30 (0.46)	29.00 (0.45)

Table 7-5: OO1 Performance Results. This table shows the elapsed time for each part of the OO1 benchmark. For each test, the reported results are the average over ten tests with the standard deviations in parentheses. As the access pattern is non-sequential, all systems offer virtually identical performance.

7-5 shows the results for the three file systems on the OO1 benchmark.

All the file systems perform comparably on this benchmark. Since there is no sequentiality to the access patterns, the contiguous layout of EFS and LFS provide no benefit. While we might expect LFS to demonstrate superior performance on the insert test, there is virtually no difference from EFS and FFS because nearly all the writes (97%) are asynchronous.

The cleaner was running during nine out of ten iterations of the benchmark, but introduced virtually no overhead. The reason is that most of the cleaning was the result of removing and recreating the database on each iteration. Therefore most segments to be cleaned were empty (88%) and those that contained live data averaged 20% utilization.

Field Name	Description
unique1	integer; unique keys, nonclustered
unique2	integer; unique keys, clustered (PRIMARY KEY)
two	integer; 2 unique values
four	integer; 4 unique values
ten	integer; 10 unique values
twenty	integer; 20 unique values
hundred	integer; 100 unique values, used for nonunique, nonclustered index
thousand	integer; 1000 unique values
twothous	integer; 2000 unique values
fivethous	integer; 5000 unique values
tenthous	integer; 10000 unique values
odd100	integer; odd values 1-99
even100	integer; even values 2-100
stringu1	char(52); unique keys, nonclustered
stringu2	char(52); unique keys, clustered (ALTERNATE PRIMARY KEY)
string4	char(52); four unique values

Table 7-6: Relation Attributes for the Wisconsin Benchmark.

7.7. The Wisconsin Benchmark

The next database benchmark is the Wisconsin Benchmark described in [BITT83] and [DEWI91]. The goal of this test is to consider the file systems in terms of their ability to support complex query processing workloads. The test database consists of four relations: ONEKTUP, a relation with 1000 tuples, TENKTUP1 and TENKTUP2, each with 10,000 tuples, and BPrime which is 10% of TENKTUP2. Each relation contains 13 integer attributes and three 52-character attributes. The same database package used in Section 7.6 is used here and yields a total database size (including indices) of 8.5 MB. The attributes are summarized in Table 7-6.

The benchmark consists of issuing the set of queries, described in Table 7-7, against this database. The elapsed time is measured individually for each query. The results reported in Table 7-8 are the averages and standard deviations of 10 measurements for each query.

For many of the queries, the performance is similar across all the file systems as the benchmark is dominated by non-sequential reads. In fact, for most queries, reads account for over 95% of the number of I/Os issued and over 90% of the bytes transferred. The following discussion focuses on those queries for which there are noticeable differences between the file systems' performance.

On query 1, LFS demonstrates approximately 25% worse performance than either of EFS or FFS. It also exhibits fairly substantial variation in performance (the standard deviation across the runs is nearly 25% of the elapsed time). Examination of the kernel counters that monitor disk traffic shows that the creation of the temporary relation occasionally causes a segment write to take place, delaying completion of the query.

However, this same phenomenon, the bundling of writes, also explains why LFS outperforms EFS and FFS on several queries, notably, queries 3, 15, 16 and 17. Each of these queries is building a temporary relation. The relation is a B-Tree, and each time a page is evicted from the data manager's buffer pool, one or more pages are written into the kernel's cache.

The kernel disk counters indicate that EFS and FFS are performing nearly four times the number of writes as LFS. The explanation lies in the write policy for EFS and FFS. As full pages are written into the cache, they are written asynchronously in FFS and potentially clustered in EFS. Since the access pattern is non-sequential, EFS performs little, if any clustering. As the data manager's cache is very small (64 kilobytes per B-Tree), the same pages are being overwritten in the kernel's cache and are being written to disk repeatedly. In contrast, LFS keeps the entire temporary relation (200 kilobytes) in the cache and writes it only once.

Queries 30, 31, and 32 also show LFS performing from one-sixth to one-third faster. All these queries are updates that require writing one or more pages to multiple (3) files. By looking at the kernel disk counters, it is apparent that in all three queries, LFS is performing one-third the number of writes as are the other systems. Both EFS and FFS write one or more blocks to each of three files while LFS bundles all those writes together and performs a single, asynchronous I/O. These additional writes by EFS and FFS account for the performance difference.

This benchmark also highlighted some subtleties in the implementation of read-ahead for both EFS and LFS. The obvious read-ahead algorithm on a file system with contiguous layout is to issue maximal-sized asynchronous reads as soon as sequential access is detected. Unfortunately, the B-Trees for this test were built with 8-kilobyte pages and are running on a file system with 4-kilobyte pages. Therefore, every page access to the B-Tree appears to begin a sequential access (i.e. the access pattern is: 0, 1, 2, 22, 23, 41, 42, ...). If the maximal-sized read-ahead is performed, many pages are read into the cache unnecessarily.

Both EFS and LFS were initially implemented with the maximal read-ahead algorithm and the resulting performance for the queries that included a sequential scan of a B-Tree resulted in devastating performance -- an order of magnitude slower than FFS with its single-block read-ahead.

Query #	Type	Selectivity	Indexing
1	Selection	1%	No index
2	Selection	10%	No index
3	Selection	1%	Clustered index
4	Selection	10%	Clustered index
5	Selection	1%	Non-clustered index
6	Selection	10%	Non-clustered index
7	Selection	.01%	Clustered index
8	Selection	1%	Clustered
9	Join-2	10%	No index
10	Join-2	.01%	No index
11	Join-3	10%	No index
12	Join-2	10%	Clustered index
13	Join-2	.01%	Clustered index
14	Join-3	10%	Clustered index
15	Join-2	10%	Non-clustered index
16	Join-2	.01%	Non-clustered index
17	Join-3	10%	Non-clustered index
18	Projection	1%	No index
19	Projection	100%	No index
	Query Type	Group Size	Indexing
20	Aggregation-min	10000	No index
21	Aggregation-min	100	No index
22	Aggregation-sum	100	
23	Aggregation-min	10000	Clustered index
24	Aggregation-min	100	Clustered index
25	Aggregation-sum	100	Clustered index
	Query Type	N elements	Indexing
26	Insert	1	No index
27	Delete	1	No index
28	Update-key	1	No index
29	Insert	1	Clustered index
30	Delete-indexed, key	1	Clustered index
31	Update-key	1	Clustered index
32	Update-indexed, nonkey	1	Clustered index

Table 7-7: Wisconsin Benchmark Queries. This table summarizes the queries of the Wisconsin benchmark. The selectivity indicates what percentage of the database will be in the final result query. The number after the Join query type indicates the number of relations involved in the join. For aggregation queries, the Group Size indicates over how many elements the aggregation will be performed. The identifiers ‘‘min’’ and ‘‘sum’’ indicate the aggregation operator. For all updates, a single record is affected.

Query Type	Query #	Elapsed Time and Standard Deviation					
		FFS		EFS		LFS	
Selection	**1**	**11.82**	**(0.81)**	**11.28**	**(0.08)**	**14.34**	**(4.37)**
	2	11.12	(0.24)	10.98	(0.03)	11.72	(0.41)
	3	**0.39**	**(0.00)**	**0.39**	**(0.00)**	**0.28**	**(0.01)**
	4	1.65	(0.05)	1.61	(0.00)	1.60	(0.04)
	5	2.68	(0.00)	2.68	(0.02)	2.57	(0.10)
	6	2.73	(0.03)	2.71	(0.00)	2.54	(0.02)
	7	0.63	(0.13)	0.77	(0.04)	0.80	(0.00)
	8	0.39	(0.03)	0.37	(0.00)	0.39	(0.00)
Join	9	637.53	(9.52)	631.92	(0.50)	641.09	(23.78)
	10	578.84	(45.45)	548.55	(13.11)	545.33	(25.08)
	11	658.63	(11.58)	649.23	(0.29)	647.49	(14.77)
	12	13.61	(0.16)	13.64	(0.06)	13.00	(0.22)
	13	12.34	(0.56)	12.71	(0.07)	11.27	(0.09)
	14	343.48	(4.37)	346.93	(0.79)	346.89	(6.43)
	15	**48.69**	**(0.27)**	**48.83**	**(0.35)**	**38.30**	**(0.08)**
	16	**17.24**	**(0.28)**	**17.35**	**(0.11)**	**8.80**	**(0.15)**
	17	**48.91**	**(0.52)**	**49.28**	**(0.04)**	**38.68**	**(0.19)**
Projection	18	26.30	(0.09)	26.24	(0.03)	26.33	(0.26)
	19	1.67	(0.05)	1.61	(0.03)	1.52	(0.01)
Aggregation	20	10.84	(0.68)	10.27	(0.02)	10.13	(0.15)
	21	10.70	(0.37)	10.29	(0.04)	10.14	(0.14)
	22	10.53	(0.29)	10.31	(0.03)	11.03	(0.10)
	23	0.12	(0.00)	0.12	(0.00)	0.76	(0.04)
Update	24	221.94	(5.54)	216.35	(0.17)	218.67	(2.59)
	25	221.90	(5.73)	216.49	(0.26)	219.21	(2.79)
	26	0.15	(0.03)	0.15	(0.03)	0.12	(0.01)
	27	10.67	(0.46)	10.57	(0.00)	10.70	(0.14)
	28	1.92	(0.02)	1.92	(0.02)	1.79	(0.05)
	29	0.40	(0.01)	0.40	(0.00)	0.33	(0.01)
	30	**0.39**	**(0.02)**	**0.38**	**(0.00)**	**0.22**	**(0.00)**
	31	**0.48**	**(0.02)**	**0.49**	**(0.02)**	**0.35**	**(0.02)**
	32	**0.30**	**(0.02)**	**0.32**	**(0.00)**	**0.22**	**(0.01)**

Table 7-8: Elapsed Time for the Queries of the Wisconsin Benchmark. This table shows the average elapsed time and the standard deviation for the ten iterations of each query. The entries that are emboldened are queries where there were significant performance differences between the file systems. For the most part, the asynchronous write behavior of LFS and its ability to cluster writes from unrelated files explains its superior performance.

In order to address this problem, the read-ahead algorithm was modified to use a read-ahead window size that grows when sequential accesses are detected and shrinks when non-sequential accesses are detected. The read-ahead window size is initialized to 1. When sequential access is determined, the minimum of the number of contiguous blocks on disk and the window size are read asynchronously. Then the read-ahead window size is set equal to twice the length of the last read-ahead, bounded by the maximum transfer size of the device. When non-sequential access is detected, the window size is halved.

This benchmark shows LFS' superior handling of temporary files and of processing writes to multiple files. The first issue, writing temporary files efficiently, is usually relegated to the memory-based file systems [MCKU90] in BSD systems. However, when the temporary files exceed the size of memory, LFS is a good alternative. The second case, writing blocks to multiple files efficiently, has no obvious solution in either EFS or FFS.

7.8. Transaction Processing Performance

The industry standard TPC-B is used as the last of the database-oriented tests. This is the same benchmark used and described in Chapter 5. It is a modified version of the TPC-B benchmark, configured for a 10 transaction per second system. The data manager is that described in [SELT92] and the tests here are run single-user without a redundant log. Each measurement in Table 7-9 represents ten runs of 1000 transactions. The counting of transactions is not begun until the buffer pool has filled, so the measurements do not exhibit any artifacts of an empty cache. Transaction run lengths of greater than 1000 were measured, but there was no noticeable change in performance after the first 1000 transactions.

When the cleaner is not running, LFS provides a 15% performance improvement over EFS. However, the impact of the cleaner is far worse than was anticipated. TPC-B randomly updates blocks in the 237 megabyte account relation, leaving most segments fairly full. During the course of the benchmark, the cleaner cleaned approximately one segment for every 50 transactions executed. On average, the cleaned segments were 71% utilized and cleaner writes accounted for between 60% and 80% of the total blocks written and 31% of all blocks transferred.

In an attempt to reduce cleaner overhead, a second set of tests were run with a smaller segment size (256 kilobytes). The performance before cleaning is the same as for the one-megabyte case, but the after-cleaning performance is only slightly better (about 6%). As in the one-megabyte case, the majority of the writes performed are on behalf of the cleaner (60-70%). While the smaller segment size reduces the variation in response time as evidenced through the smaller standard deviation, it does not significantly improve performance as most of the write activity is due to the cleaner.

The cleaner impact in Chapter 5 was much less dramatic because the file system had more free space, and the cleaner didn't run as frequently. For the tests presented here, the disk was

	Transactions per second	Elapsed Time 1000 transactions
FFS	14.2	70.23 (1.0%)
EFS	16.8	59.51 (2.1%)
LFS (no cleaner)	19.3	51.75 (0.6%)
LFS (cleaner, 1M)	11.6	85.86 (5.3%)
LFS (cleaner, 256 K)	12.4	80.72 (1.8%)

Table 7-9: TPC-B Performance Results. The TPC-B database was scaled for a 10 transaction-per-second system (1,000,000 accounts, 100 tellers, and 10 branches). The elapsed time and standard deviation, as a percent of the elapsed time, is reported for runs of 1000 transactions. The LFS results show performance before the cleaner begins to run and after the cleaner begins to run. The after-cleaner performance is shown both for a file system with one-megabyte segments and one with 256 kilobyte segments.

running at 80% utilization as opposed to the 70% utilization in Chapter 5. This indicates that log-structured file systems are particularly sensitive to the disk capacity. Although a user-level cleaner avoids synchronization costs between user processes and the kernel, it cannot avoid contention on the disk arm.

7.9. Super-Computer Benchmark

To evaluate the file systems' performance on super-computing applications, this benchmark simulates the I/O behavior traced in [MILL91]. The tracing study analyzed seven scientific applications. Table 7-10 summarizes the data provided for four representative applications. As the tracing data did not provide detailed information on how the I/O accesses corresponded to files or on the sizes of the individual I/Os, this benchmark derives and uses the average read and write sizes to simulate a supercomputer workload.

For each application, the average read and write sizes are derived by solving the following set of equations:

$$(1) \qquad RW = \frac{\bar{R}N_r}{\bar{W}N_w}$$

and

$$(2) \qquad \bar{R}N_r + \bar{W}N_w = (N_r + N_w)A$$

Where:

RW is the read to write ratio in bytes,
$\bar{R}$ is the average read size,
$\bar{W}$ is the average write size,
N_r is the number of reads,
N_w is the number of writes, and
A is the average I/O size.

Solving for $\bar{R}$ in equation (1) yields:

$$\bar{R} = \frac{RW(\bar{W}N_w)}{N_r}$$

Application	Total Data Size	Average I/O Size (in kilobytes)	Number of reads	Number of writes	Data Read/Write Ratio
bvi	171.0	16.1	913	185	2.31
ccm	11.6	31.9	135	128	1.07
les	224.0	324.0	74	81	0.95
venus	55.2	456.0	60	32	1.80

Table 7-10: Supercomputer Applications I/O Characteristics. These characteristics were obtained from real super-computer workload traces and are used here to simulate a super-computer I/O workload. Column 6 indicates the read-write ratio in terms of the total number of bytes accessed. The read-write ratio in terms of number of operations can be derived from columns 4 and 5.

Substituting $\bar{R}$ in equation (2) yields:

$$\frac{\bar{W}N_w(1+RW)}{(N_w+N_r)} = A$$

$$\bar{W} = \frac{A(N_w + N_r)}{N_w(1+RW)}$$

Once the average read and write sizes have been established, a single file is created. The file is partitioned into average-write-sized units and these units are then written in a random order to create the file. The benchmark consists of opening the file and performing 1000 file read and write requests distributed according to the ratios in Table 7-10. All reads and writes are the same size ($\bar{R}$ and $\bar{W}$ respectively), but the location within the file of each read or write is selected randomly.

Table 7-11 shows the results of running each of the four simulated workloads on the three file systems. The average I/O size is smallest for the application at the top of the table and increases down the table to VENUS with the largest average access. Similarly, the resulting throughput improves for all three systems.

Both EFS and LFS demonstrate the ability to perform large, sequential I/O well, achieving nearly the same performance they achieved for the raw file system performance benchmark in Section 7.3. As expected, LFS offers the best write performance across all but one of the programs, with performance ranging from 35% to 100% better than that of the other systems.

The improvement due to LFS is largest when the I/Os are small and the ratio of read to write requests low. In the BVI benchmark, the I/Os are small but the read to write ratio is nearly 5 to 1. Therefore, LFS offers a substantial improvement (nearly 50%) but not nearly as impressive as the improvement for CCM where the read to write ratio is nearly 1 to 1. The higher read to write ratio means that LFS is more likely to have to perform a seek before each write request and therefore doesn't derive the maximum benefit from its sequential writing.

Since LFS and EFS are using the same read-ahead policy, the read performance is a function of how successful the systems are at contiguous allocation and how much LFS cleaning overrides the contiguous allocation. There are two ways in which blocks become eligible for reclamation in this test. First, the writes issued overwrite existing data leaving partially filled segments. Second, the data file is deleted between each test resulting in a large number of empty segments. The statistics from the cleaner reveal that 78% of the segments cleaned are empty and of those non-empty, their average utilization is 29%. As a result, the cleaner impact is minimal. Only 2% of all blocks written are from the cleaner and only 1% of all I/Os are from the cleaner.

In the BVI, CCM, and VENUS benchmarks, LFS read performance is very close to EFS read performance (5% worse for BVI, 8% and 9% better for CCM and VENUS). The only substantial difference in read performance is in the LES benchmark (LFS is 23% slower than EFS). Closer inspection of the cleaning statistics reveals that most cleaning occurs during this benchmark, because there are more writes than reads and the writes are very large. In fact, of the 150 segments cleaned, 148 take place during this benchmark.

This benchmark demonstrates that both LFS and EFS can deliver a large fraction of the available I/O bandwidth to applications if the I/Os are large. When they are small, LFS performs moderately better as long as the cleaning required is not substantial.

7.10. Conclusions

This chapter has presented a variety of benchmark programs. As expected, LFS demonstrates superior write performance in most environments. One unexpected result is that cleaning can have dramatic effects on LFS performance, particularly when the application is overwriting a small steady stream of data, as in TPC-B. A second surprising result is the detrimental effect of

	Average I/O size		FFS	EFS	LFS
BVI Read	13.8 KB	Average time per I/O Throughput (KB/sec)	0.07 (0.02) 206.02	0.06 (0.02) 227.60	0.06 (0.02) 214.13
BVI Write	29.6 KB	Average time per I/O Throughput (KB/sec)	0.13 (0.09) 215.56	0.12 (0.09) 242.40	0.06 (0.05) 474.40
CCM Read	32.9 KB	Average time per I/O Throughput (KB/sec)	0.10 (0.02) 332.59	0.08 (0.02) 392.55	0.08 (0.02) 425.98
CCM Write	32.4 KB	Average time per I/O Throughput (KB/sec)	0.11 (0.02) 289.49	0.11 (0.03) 299.68	0.05 (0.01) 601.89
LES Read	338.6 KB	Average time per I/O Throughput (KB/sec)	0.75 (0.15) 440.72	0.37 (0.06) 900.54	0.48 (0.24) 691.11
LES Write	325.6 KB	Average time per I/O Throughput (KB/sec)	1.15 (0.07) 276.48	0.39 (0.04) 811.33	0.40 (0.09) 792.19
VENUS Read	460.2 KB	Average time per I/O Throughput (KB/sec)	0.98 (0.23) 458.39	0.43 (0.09) 1046.57	0.39 (0.06) 1152.99
VENUS Write	479.4 KB	Average time per I/O Throughput (KB/sec)	1.48 (0.12) 316.69	0.79 (0.02) 593.56	0.59 (0.03) 798.61

Table 7-11: Performance of the Supercomputer Benchmark. The average I/O size, shown in the second column, increases down the table as does the resulting performance. The average times per I/O are reported to show the variations in response time exhibited by the different systems. LFS offers superior write performance on three of the four programs and superior read performance on two of them. For the applications with the largest I/Os (LES and VENUS), both read and write performance approaches that of the benchmarks in Section 7.3.

aggressive read-ahead policies on workloads that appear sequential, but are not, as was discussed in the Wisconsin benchmark in Section 7.7.

Chapter 8

Conclusions

This thesis makes three main research contributions. It offers a fundamental understanding of how file system allocation affects the performance of real applications. Second, it highlights many key design issues involved in building log-structured file systems and offers a design that can function in constrained environments. Finally, it demonstrates that adding transactions to a file system does not have to impact performance and can offer an attractive alternative to user-level transaction management.

Several different allocation policies have been described and evaluated by simulation and implementation. As expected, the file system performance was extremely workload dependent. Micro benchmarks, such as the raw performance tests of Section 7.3, do not capture the real behavior of file systems, and single-user macro benchmarks often do not provide insight into multi-user performance. However, file systems which favor contiguous disk allocation (the log-structured system and the extent-based system) uniformly provide the best performance, although LFS is penalized in performance due to the presence of the cleaner.

8.1. Chapter Summaries

Chapter 3 examined the use of variable sized disk blocks to improve a disk system's bandwidth, and concluded that policies which use large allocations to improve sequential performance can do so without hindering the performance of small file accesses.

Chapter 4 focused on an online transaction processing workload which did not benefit from contiguous allocation. This simulation concluded that log-structured file systems offer the potential for improved performance in these environments. It also showed that in many cases an embedded transaction manager provides comparable performance to user-level data managers.

Chapter 5 used an implementation study to corroborate the conclusions from the simulation of Chapter 4. However, the realities of implementation, particularly perturbations in performance due to the cleaner, dispute the simulation results. Rather than corroborating the simulation results, it indicated areas of improvement in the implementation of log-structured file systems.

In Chapter 6, a new design of LFS was presented that addressed most of the issues raised in Chapter 5, as well as many others. The new implementation has the cleaner running in user-space, does not reserve large amounts of kernel memory, and shares most of its code with the Fast File System.

Chapter 7 compared a range of allocation policies across diverse workloads. These results confirmed the simulation results of Chapter 3, in terms of the performance of read-optimized file systems. Once again, transaction processing workloads performed substantially worse than predicted in Chapter 3. Furthermore, the cleaning overhead of LFS can impact performance substantially when empty segments are not available for reclamation.

Although the workloads in Chapter 7 came from vastly different environments (software development, query processing, online transaction processing, and super-computing), the extent-based file system offered the best or nearly the best performance on every test. However, all the tests, except for the multi-user Andrew benchmark, were single-user tests. As a result, LFS was unable to obtain the maximum benefit from its write-optimization.

While LFS does offer superior write performance under appropriate workloads, the cleaning penalty can be quite high. The measurements in [ROSE92] indicate that LFS achieves 70% write performance, however this was a statically computed write cost averaged over the lifetime of the file system, and does not reflect the performance perturbations when the cleaner runs or the response time observed by applications. If the disk system is utilizing a large fraction of its disk bandwidth for the cleaner, as was the case in Section 7.8, the impact is unacceptable. When LFS is performing best, that is, bundling many writes into a single, contiguous write, the disk system is either becoming full or existing data is being overwritten. In the latter case, the cleaner is required to run and the impact on applications running at that time can be severe.

A second observation is the impact of segment size on the application response time. Using large segments in the transaction processing environment resulted in a performance penalty of 40% when the cleaner ran. Reducing the segment size only reduced the penalty to 35%. In this workload, the file system does not obtain much benefit from batching a large number of writes because the test is running single-user and each transaction is issuing a small number of writes. Therefore, although LFS can perform better on this workload when the segment size is small, both FFS and EFS do much better in this environment because they aren't competing with the cleaner for the disk arm.

Managing read-ahead in both EFS and LFS is tricky. The pitfalls became apparent on the initial measurements for the Wisconsin benchmark results discussed in Section 7.7. When the file system performed read-ahead in maximum-sized units, access patterns which appeared sequential, but were not, resulted in cache thrashing and some queries showed an order of magnitude slow down as a result.

While using a read-ahead window size fixed the problem for this benchmark, there are undoubtedly workloads for which that solution is still unsatisfactory. The better solution is to place read-ahead blocks toward the least-recently-used end of the LRU queue. Currently, when blocks return from I/O successfully, they are either placed at the most-recently-used end of the LRU queue or the AGE queue. Buffers on the AGE queue are immediately eligible for reclamation, so placing read-ahead blocks there doesn't work. Instead, they should be placed far enough away from the least-recently-used end of the the LRU queue so that if the access pattern is genuinely sequential, they will still be in the cache when requested.

LFS does provide some compelling benefits over the other systems. Adding transactions to a log-structured file system is easy and provides a mechanism not offered by the other systems. The ''no-overwrite'' policy in LFS makes restoring deleted files simple. Versioning and historical archiving are obvious extensions to this functionality. The fast recovery of LFS also makes it a desirable alternative, as does the potential for on-line backups. Since the cleaner already reads segments to reclaim them, it could also write the segments to a backup device. Requiring the cleaner to touch every segment during a backup would result in a complete snapshot of the system.

Another domain where LFS is extremely attractive is in the high-speed networking arena. With gigabit networks just around the corner, file systems need the ability to sink large quantities of data quickly.

LFS also seems attractive for RAID [PATT88] where small writes are penalized due to parity calculation. However, empirical evidence in [BAKER92] shows that most segments are still small, so this benefit may be unattainable. Additionally, simple, extent-based systems like EFS can also take advantage of large writes to avoid the overhead of parity update.

8.1. Future Research Directions

This thesis raises at least as many new questions as it answers. The studies in Chapters 3 and 4 indicate that transactions embedded in an LFS provide some benefits. It would be interesting to add transaction support to the new BSD-LFS and exercise it thoroughly. The segment batching mechanism already present makes this an even simpler implementation than the one described in Chapter 4. The presence of segment batching raises the possibility of potentially replacing special purpose kernel mechanisms for ordering and atomicity with this general-purpose mechanism. Also, it will be interesting to consider what existing applications might benefit from native transaction support, and what new applications will become possible.

Another interesting question that arises is the application of transaction semantics to a file system. Incorporating transaction semantics into a UNIX file system is complex. Since regular UNIX utilities must continue to function normally, how should accesses to a transaction-protected file outside the context of a transaction be handled? Also, long-running transactions that write data have the potential to fill the disk system. This is discussed in the context of Quicksilver in [SCHM91], and the conclusion is that such long running transactions should be decomposed into more transactions of shorter duration.

A related question is whether LFS is viable for supporting database applications. While a conventional UNIX file system imposes a layer of address indirection above and beyond that created by a data manager, LFS adds another level of indirection by virtue of the inode map, and also provides an alternate representation in the segment summaries. Does this change the way a database interacts with the file system? The segment summary blocks provide an alternate method of reading for large files that need to be read in their entirety, but not sequentially, A reader begins at one edge of the disk, reads the segment summary block and returns any ‘‘live’’ blocks in the segment from the requested file. This could prove more efficient than reading the file in its logical block order.

Chapter 5 suggests implementing a variety of cleaning algorithms in a log-structured file system. The current cleaner’s only function is to reclaim disk space. It would be easy to envision a more intelligent cleaner that coalesced large files or placed frequently access files in the center of the disk. However, as the cleaner already impacts performance substantially, such policies would have to make an effort to avoid unnecessary I/O.

A second approach to improving locality for large files would require investigating the segment writer’s policy. Currently, only dirty data is written to disk during segment writing. However, if there are clean blocks in the cache that would restore a file’s contiguous layout, it might be beneficial to write the clean blocks as well.

8.2. Summary

LFS is an exciting vehicle for extending the functionality of the file system. It is an excellent framework in which to provide embedded transaction support, versioning, and archival. However, its performance in certain applications, particularly transaction processing, is severely limited by its ability to perform garbage collection. As a result, for most workloads analyzed in this thesis, conventional file systems that provide contiguous on-disk layout are the more attractive alternative.

References

[ACCE86] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., ‘‘MACH: A New Kernel Foundation for UNIX Development,’’ *Proceedings of the 1986 Summer Usenix*, Atlanta, GA, June 1986, 93-113.

[ADAP85] *SCSI User Guide*, Adaptive Data Systems Inc., Pomona, CA, 1985.

[ANDE91] Anderson, T., Bershad, B., Lazowska, E., Levy, H., ‘‘Scheduler Activations: Effective Kernel Support for User Level Management of Parallelism,’’ *Proceedings of the Thirteenth Symposium on Operating System Principles*, Monterey, CA, October 1991, 95-109. Published as *Operating System Review 25*, 5 (October 1991).

[ANDR89] Andrade, J., Carges, M., Kovach, K., ‘‘Building an On-Line Transaction Processing System On UNIX System V,’’ *CommUNIXations*, November/December 1989.

[ANON85] Anon et. al., ‘‘A Measure of Transaction Processing Power,’’ *Datamation*, April 1985.

[ARAL89] Aral, Z., Gertner, I., Langerman, A., Schaffer, G., Bloom, J., Doeppner, T., ‘‘Variable Weight Processes with Flexible Shared Resources,’’ *Proceedings of the 1989 Winter Usenix*, San Diego, CA, February 1989, 405-412.

[ASTR76] Astrahan, M., Blasgen, M., Chamberlain, K., Eswaran, K., Gray, J., Griffiths, P., King, W., Traiger, I., Wade, B., Watson, V., ‘‘System R: Relational Approach to Database Management,’’ *ACM Transactions on Database Systems 1*, 2 (1976), 97-137.

[BAKER91] Baker, M., Hartman., J., Kupfer., M., Shirriff, L., Ousterhout, J., ‘‘Measurements of a Distributed File System,’’ *Proceedings of the 13th Symposium on Operating System Principles*, Monterey, CA, October 1991, 198-212. Published as *Operating Systems Review 25*, 5 (October 1991).

[BAKER92] Baker, M., Asami, S., Deprit, E., Ousterhout, S., Seltzer, M., ‘‘Non-Volatile Memory for Fast, Reliable File Systems,’’ to appear in *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[BAYER77] Bayer, R., Scholnick, M., ‘‘Concurrency Operations on B-Trees,’’ *Acta Informatica*, 1977.

[BART81] Bartlett, J., ‘‘A NonStop Kernel,’’ Tandem Computers, Technical Report 81.4, PN87603, June 1981.

[BEN69] Bensoussan, A., Clingen, C., Daley, R., ‘‘The Multics virtual memory,’’ *Proceedings of the Second Symposium on Operating Systems Principles*, Princeton University, October 1969, 30-42. Published as *Operating Systems Review 13*, 5 (October 1987).

[BERN80] Bernstein, P., Goodman, N., ‘‘Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems,’’ *Proceedings 6th International Conference on Very Large Data Bases*, October 1980.

[BERS92] Bershad, B., Redell, D., Ellis, J., ‘‘Fast Mutual Exclusion for Uniprocessors,’’ to appear in *Proceedings of ASPLOS-V*, Boston, MA, October 1992.

[BITT83] Bitton, D., DeWitt, D., Turbyfill, C., ‘‘Benchmarking database systems: A systematic approach,’’ Proceedings of the Ninth Conference on Very Large Data Bases, 1983,

[BORR90] Borr, A., ‘‘Guardian 90: A Distributed Operating System Optimized Simultaneously for High-Performance OLTP, Parallelized Batch/Query, and Mixed Workloads,’’ Tandem Computers, Technical Report 90.8, 49788, July 1990.

[BSD91] DB(3), *4.4BSD Unix Programmer's Manual Reference Guide*, University of California, Berkeley, 1991.

[CAR92] Carson, S., Setia, S., ‘‘Optimal Write Batch Size in Log-structured File Systems,’’ *Proceedings of 1992 Usenix Workshop on File Systems*, Ann Arbor, MI, May 21-22 1992, 79-91.

[CATT91] Cattell, R.G.G., ‘‘An Engineering Database Benchmark,’’ *The Benchmark Handbook for Database and Transaction Processing Systems*, J. Gray, editor, Morgan Kaufman, 1991, 247-280.

[CHAM81] Chamberlain, D., et. al., ‘‘A History and Evaluation of System R,’’ *Communications of the ACM 24*, 10 (October 1981), 632-646.

[CHAN88] Chang, A., Mergen, M., ‘‘801 Storage Architecture and Programming,’’ *ACM Transactions on Computer Systems 6*, 1 (February 1988), 28-50.

[CHER88] Cheriton, D., ‘‘The V Distributed System,’’ *Communications of the ACM 31*, 3 (March 1988), 314-333.

[DEWI84] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., Wood, D., ‘‘Implementation Techniques for Main Memory Database Systems,’’ *Proceedings of SIGMOD*, June 1984, 1-8.

[DEWI91] DeWitt, D., ‘‘The Wisconsin Benchmark: Past, Present, and Future,’’ *The Benchmark Handbook for Database and Transaction Processing Systems*, J. Gray, editor, Morgan Kaufman, 1991, 119-166.

[DUBO82] DuBourdieux, D., ‘‘Implementation of Distributed Transactions,’’ Proceedings of the Sixth Berkeley Workshop on Distributed Data Bases and Computer Networks, Asilomar, CA, February 1982.

[ELKH84] Elkhardt, K., Bayer, R., ‘‘A Database Cache for High Performance and Fast Restart in Database Systems,’’ *ACM Transactions on Database Systems 9*, 4 (December 1984), 503-525.

[FIN87] Finlayson, R., Cheriton, D., ‘‘Log Files: An Extended File Service Exploiting Write-Once Storage,’’ *Proceedings of the Eleventh Symposium on Operating Systems Principles*, Austin, TX, November 1987, 139-148. Published as *Operating Systems Review 21*, 5 (November 1987).

[FUJI84] M2361A Mini-Disk Drive Engineering Specifications, Fujitsu Limited, 1984.

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., ‘‘Granularity of locks and degrees of consistency in a large shared data base,’’ *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, 365-394.

[HAER83] Haerder, T. Reuter, A. ‘‘Principles of Transaction-Oriented Database Recovery,’’ *Computing Surveys 15*, 4 (1983), 237-318.

[HASK88] Haskin, R., Malachi, Y., Sawdon, W., Chan, G., ‘‘Recovery Management in Quicksilver,’’ *ACM Transactions on Computer Systems 6*, 1 (February 1988), 82-108.

[HELL89] Helland, P., ‘‘The TMF Application Programming Interface,’’ Tandem Computers, Technical Report 89.3, 21680, February 1989.

[HOWA88] Howard, J., Kazar, Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, N., West, M., ‘‘Scale and Performance in a Distributed File System,’’ *ACM Transaction on Computer Systems 6*, 1 (February 1988), 51-81.

[IBM] *MVS/XA JCL User’s Guide*, International Business Machines Corporation, chapter 15, 15-29.

[IBM80] *IMS/VS Version 1 General Information Manual*, GH20-1260, IBM Corporation, White Plains, NY, September 1980.

[KAZ90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, A., Tu, S., Zayas, E., ‘‘DECorum File System Architectural Overview,’’ *Proceedings of the 1990 Summer Usenix* Anaheim, CA, June 1990, 151-164.

[KLEI86] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Usenix Conference Proceedings*, June 1986, 238-247.

[KOCH87] Philip D. L. Koch, ‘‘Disk File Allocation Based on the Buddy System,’’ *ACM Transactions on Computer Systems, 5*, 4 (November 1987), 352-370.

[KOND92] Kondoff, A., Hewlett-Packard Company, Private Conversation, March 1992.

[KNOW65] Knowlton, K.D., "A Fast Storage Allocator," *Communications of the ACM 8*, 10 (October 1965), 623-625.

[KNUT69] Knuth, D., *The Art of Computer Programming,*Vol 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1969, 442-445.

[KUM87] Kumar, A., Stonebraker, M., ‘‘Performance Evaluation of an Operating System Transaction Manager,’’ *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, England, 473-481.

[KUM89] Kumar, A., Stonebraker, M., ‘‘Performance Considerations for an Operating System Transaction Manager,’’ *IEEE Transactions on Software Engineering 15*, 6 (June 1989), 705-714.

[KUNG81] Kung, H. T., Richardson, J., ‘‘On Optimistic Methods for Concurrency Control,’’ *ACM Transactions on Database Systems 6* 2 (June 1981), 213-226.

[LEHM81] Lehman, P., Yao, S., ‘‘Efficient Locking for Concurrent Operations on B-trees,’’ *ACM Transactions on Database Systems 6*, 4 (December 1981).

[LISK83] Liskov, B., Scheifler, R., ‘‘Guardians and actions: Linguistic support for robust, distributed programs,’’ *ACM Transactions on Programming Language Systems 5* 3 (July, 1983), 381-404.

[MCKU84] McKusick, M., Joy, W., Leffler, S., Fabry, R., ‘‘A Fast File System for UNIX,’’ *Transactions on Computer Systems 2,* 3 (August 1984), 181-197.

[MCKU86] McKusick, M., Karels, M., ‘‘A New Virtual Memory Implementation for Berkeley UNIX,’’ Computer Systems Research Group, University of California, Berkeley, CA, 1986.

[MCKU90] McKusick, M., Karels, M., Bostic, K., ‘‘A Pageable Memory Based Filesystem,’’ *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 137-144.

[MCVO91] McVoy, L, Kleiman, S., ‘‘Extent-like Performance from a UNIX File System,’’ Proceedings of the 1991 Winter Usenix, Dallas, TX, January 1991, 33-44.

[MILL91] Miller, E., ‘‘Input/Output Behavior of Supercomputing Applications,’’ Technical Report CSD-91-616, Dept. of Computer Science, Univ of California, Berkeley, December 1991.

[MITC82] Mitchell, J., Dion, J., ‘‘A Comparison of Two Network-Based File Servers,’’ *Communications of the ACM, 25* 4 (April 1982), 233-245.

[MUEL83] Mueller, E. etc al., ‘‘A Nested Transaction Mechanism for LOCUS,’’ *Proceedings Ninth Symposium on Operating System Principles*, October 1983, 71-89. Published as *Operating Systems Review 17*, 5 (October 1983).

[ORA89] *Oracle Database Administrator’s Guide*, Oracle Corporation, 3601-V6.0, April 1989.

[OUST85] Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J., ‘‘A Trace-Driven Analysis of the UNIX 4.2BSD File System,’’ *Proceedings of the Tenth Symposium on Operating System Principles*, December 1985, 15-24. Published as *Operating Systems Review 19*, 5 (December 1985).

[OUST88] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., Welch, B., ‘‘The Sprite Network Operating System,’’ *IEEE Computer 21*, 2 (February 1988), 23-36.

[OUST89] Ousterhout, J., Douglis, F., ‘‘Beating the I/O Bottleneck: A Case for Log-structured File Systems,’’ *Operating Systems Review 23*, 1, January 1989, 11-27. Also published as UCB. technical report UCB/CSD 88/467.

[PATT88] Patterson, D., Gibson, G., Katz, R., ‘‘A Case for Redundant Arrays of Inexpensive Disks (RAID),’’ *Proceedings of SIGMOD*, Chicago, IL, June 1988, 109-116.

[PU86] Pu, C., Noe, J., ‘‘Design of Nested Transactions in Eden,’’ Technical Report 85-12-03, Dept. of Computer Science, Univ of Washington, Seattle, WA, 1986.

[ROSE90] Rosenblum, M., Ousterhout, J., ‘The LFS Storage Manager,’’ *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 315-324.

[ROSE91] Rosenblum, M., Ousterhout, J. K., ''The Design and Implementation of a Log-Structured File System,'' *Proceedings of the Symposium on Operating System Principles*, Monterey, CA, October 1991, 1-15. Published as *Operating Systems Review 25*, 5 (October 1991). Also available in *Transactions on Computer Systems 10*, 1 (February 1992), 26-52.

[ROSE92] Rosenblum, M., ''The Design and Implementation of a Log-structured File System,'' PhD Thesis, University of California, Berkeley, June 1992. Also available as Technical Report UCB/CSD 92/696.

[RTI83] Relational Technology, Inc., *INGRES Reference Manual*, 1983.

[SCHL90] Schloss, G., Stonebraker, M., ''Distributed RAID -- A New Multiple Copy Algorithm,'' *Proceedings 6th Annual International Conference on Data Engineering*, April 1990.

[SCHM91] Schmuck, F., Wyllie, J., ''Experience with Transactions in QuickSilver,'' *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, Monterey, CA, October 1991, 239-253. Published as *Operating Systems Review 25*, 5 (October 1991).

[SELT90] Seltzer, M., Chen, P., Ousterhout, J., ''Disk Scheduling Revisited,'' *Proceedings of the 1990 Winter Usenix*, Washington, D.C., January 1990, 313-324.

[SELT91] Seltzer, M., Stonebraker, M., ''Read Optimized File Systems: A Performance Evaluation,'' *Proceedings 7th Annual International Conference on Data Engineering*, Kobe, Japan, April 1991, 602-611.

[SELT92] Seltzer, M., Olson, M., ''LIBTP: Portable, Modular Transactions for UNIX,'' *Proceedings of the 1992 Winter Usenix*, San Francisco, CA, January 1992, 9-25.

[SPE88A] Spector, Rausch, Bruell, ''Camelot: A Flexible, Distributed Transaction Processing System,'' *Proceedings of Spring COMPCON 1988*, February 1988, 432-437.

[SPE88B] Spector, A, Swedlow, K., *Guide to the Camelot Distributed Transaction Facility*, Computer Science Department, Carnegie-Mellon University, Release 1, edition 0.98(51), May 1988.

[STON81] Stonebraker, M., ''Operating System Support for Database Management,'' *Communications of the ACM 24* 7 (July 1981), 412-418.

[STON85] Stonebraker, M., ''Problems in Supporting Data Base Transactions in an Operating System Transaction Manager,'' *Operating System Review 19* 1 (January 1985), 6-14.

[STON89] Stonebraker, M., Aoki, P., Seltzer, M., ''Parallelism in XPRS,'' Electronics Research Laboratory, University of California, Berkeley, CA, Report M89/16, February 1989.

[STRA89] *VOS Transaction Processing Facility Guide*, Stratus Computer, Inc., R215-00, VOS Release 9.0, November 1989, 1:1-1:14, 4:1-4:30.

[SULL92] Sullivan, M., Olson, M., ''An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System,'' *Proceedings 8th Annual International Conference on Data Engineering*, Tempe, Arizona, February 1992.

[SULL91] Sullivan, M., Chillarege, R., ''Software Defects and Their Impact on System

Availability -- A Study of Field Failures in Operating Systems,'' *Digest 21st International Symposium on Fault Tolerant Computing*, June 1991.

[SYB90] *Sybase Administration Guide*, Sybase Corporation, 3250-4.2-Rev. 3 May 1990.

[TPCB90] Transaction Processing Performance Council, ''TPC Benchmark B Standard Specification,'' Waterside Associates, Fremont, CA., August 1990.

[THOM78] Thompson, K., ''Unix Implementation,'' *Bell Systems Technical Journal*, 57(6), part 2, July-August 1978, 1931-1946.

[TRA82] Traiger, I., ''Virtual Memory Management for Data Base Systems,'' *Operating System Review 16* 4 (October 1982), 26-48.

[WALK83] Walker, Popek, English, Kline, Thie, ''The LOCUS Distributed Operating System,'' *Proceedings 9th Symposium on Operating System Principles*, October 1983, 49-70. Published as *Operating Systems Review 17*, 5 (October 1983).

[WRI91] Wright, R., ''Automatic Generation of Synthetic Disk Traces,'' Hewlett-Packard Laboratories, HPL-CSP-91-16, July 1991.