Transaction Support in a Log-Structured File System

Margo I. Seltzer

Harvard University Division of Applied Sciences

Abstract

This paper presents the design and implementation of a transaction manager embedded in a log-structured file system [11]. Measurements show that transaction support on a log-structured file system offers a 10% performance improvement over transaction support on a conventional, read-optimized file system. When the transaction manager is embedded in the log-structured file system, the resulting performance is comparable to that of a more traditional, user-level system. The performance results also indicate that embedding transactions in the file system need not impact the performance of non-transaction applications.

1. Introduction

Traditionally, transaction support has been provided by database management systems running as user-level processes. These systems maintain their own cache of frequently accessed data pages and perform their own scheduling, allowing multiple processes to access a single database. When transaction support is provided as an operating system service, concurrency control and crash recovery become transparently available to all applications instead of only to the clients of the database manager. Additionally, such an environment provides a single recovery paradigm (transaction recovery) rather than two separate recovery paradigms (file system recovery and database recovery).

The performance of user-level transaction support and embedded transaction support is compared in the simulation study presented in [14]. The study concluded that a log-structured file system (LFS) offers better performance than a read-optimized file system for a short-transaction workload, and that the embedded transaction manager performs as well as a user-level transaction manager except in extremely contentious environments. These results were based solely on simulation and did not consider the functional redundancy that results when transaction management is done in user-space. This paper addresses these issues.

The remainder of this paper is organized as follows. First, the log-structured file system is briefly described. Next, the user-level transaction model and implementation and the embedded transaction model and implementation are presented. Finally, performance results are discussed.

2. A log-structured file system

A log-structured file system, described fully in [12], is a hybrid between a simple, sequential database log, and a UNIX[1] file system [8]. Like a database log, all data is written sequentially. Like a UNIX file system, a file's physical disk layout is described by an index structure (**inode**) that contains the disk addresses of some number of **direct**, **indirect**, and **doubly indirect** blocks. **Direct blocks** contain data, while **indirect blocks** contain disk addresses of **direct blocks**, and **doubly indirect blocks** contain disk addresses of **indirect blocks**. For the remainder of this paper, the index structures and both single and double indirect blocks are referred to as **meta-data**.

While conventional UNIX file systems preallocate disk space to optimize sequential reading, an LFS allocates disk space dynamically to optimize the write performance of the file system. In a conventional file system, the blocks within a file are assigned permanent disk addresses, and each time a block is modified, the same disk block is overwritten. An LFS does not assign permanent disk addresses to blocks. Instead, when blocks are written to disk, a large number of dirty blocks, the meta-data describing them, and a summary block describing the blocks are sorted and written sequentially in a single unit, called a segment [10]. Figure 1 shows the allocation and modification of three files in a log-structured file system.

Since the structure described is an append-only log, the disk system will eventually become full, requiring a mechanism to reclaim previously used space. If there are

The work described here was supported in part by the National Science Foundation under grant MIP-8715235 and NSF IRI-9107455 and the Defense Advanced Research Projects Agency under contract DABT63-92-C-0007.

[1] UNIX is a trademark of USL.

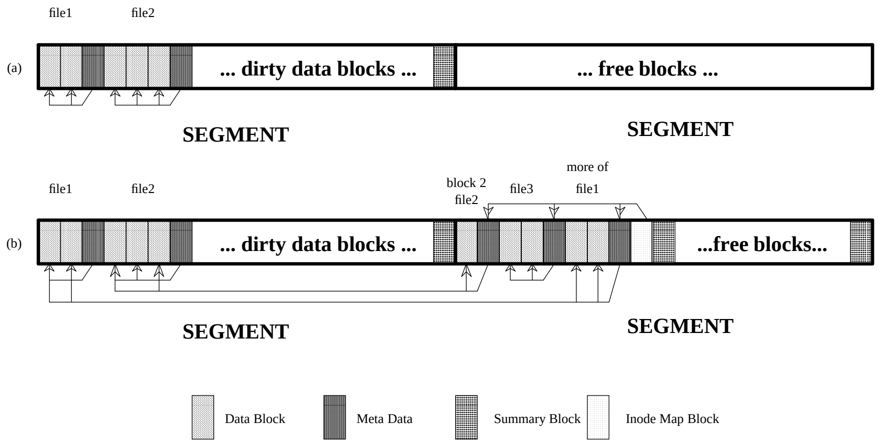

Figure 1: A Log-Structured File System. In figure (a), the first segment contains two new files, file1 and file2, as well as other dirty data blocks. The meta-data block following each file contains that file's index structure. In figure (b), the middle block of file2 has been modified. A new version 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 the addresses of the inodes, is written.**

files that have been deleted or modified, some blocks in the log will be ''dead'' (those that belong to deleted files or have been superseeded by later versions). The **cleaner**, a garbage collector, 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 [11].

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 similar to the database cache discussed in [4]. The database cache technique writes pages sequentially to a cache, typically on disk. As pages in this cache need to be replaced, they are written back to the actual database, which resides on a conventional file system. In a heavily used database, writing from the cache to the database can still limit performance. In a log-structured file system, the sequential log is the only representation of the database. Therefore, a log-structured file system always writes at sequential disk speeds approaching 100% of the disk bandwidth, while the database cache technique can, at best, sort I/Os before issuing writes from the cache to the disk. Simulation results show that even well-ordered writes are unlikely to achieve utilization beyond 40% of the disk bandwidth [13].

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. Therefore, no separate log file is required to hold before-images of updated records.

3. A user-level transaction system

For the experiments described in this paper, a traditional transaction system using write-ahead logging (WAL) and two-phase locking [5] was implemented to provide a basis for comparison to LFS-embedded support. The transaction application uses the record-oriented subroutine interface provided by the 4.4BSD database access routines [2] to read and write B-Tree, hashed, or fixed-length records. This record library makes calls to the buffer manager, lock manager, and log manager provided in the LIBTP library [15]. LIBTP uses the conventional transaction architecture depicted in Figure 2.

The system uses before-image and after-image logging to support both ''redo'' and ''undo'' recovery. Concurrency control is provided by a general purpose lock manager (single writer, multiple readers) which performs two-phase, page-level locking and high concurrency B-Tree locking [7]. To reduce disk traffic, the system

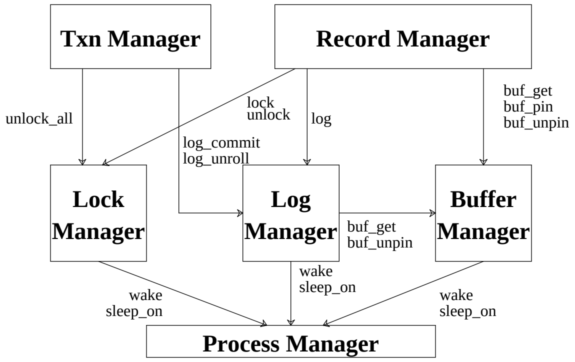

Figure 2: Conventional, User-Level Architecture.

maintains a least-recently-used (LRU) buffer cache of database pages in shared memory. Since transactions must occasionally wait for locks or buffers, a process management module is responsible for scheduling and descheduling processes. Finally, a subroutine interface is provided to begin, commit, and abort transactions.

4. The embedded implementation

In this model, transaction support is implemented within the file system. Like protections or access control lists, transaction-protection is considered to be an attribute of a file and may be turned on or off through a provided utility. The interface to transaction-protected files is identical to the interface to unprotected files (*open*, *close*, *read*, and *write*). In addition, transaction begin, commit, and abort are provided by three new system calls *txn_begin*, *txn_abort* and *txn_commit* which have no affect on unprotected files.

When transactions are embedded in the file system, the operating system's buffer cache obviates the need for a 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 flushing all dirty pages at commit guarantees the existence of after-images in the file system. Therefore, the only functionality which needs to be added to the kernel is lock management and transaction management. This architecture is depicted in Figure 3.

4.1. Data Structures and Modifications

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

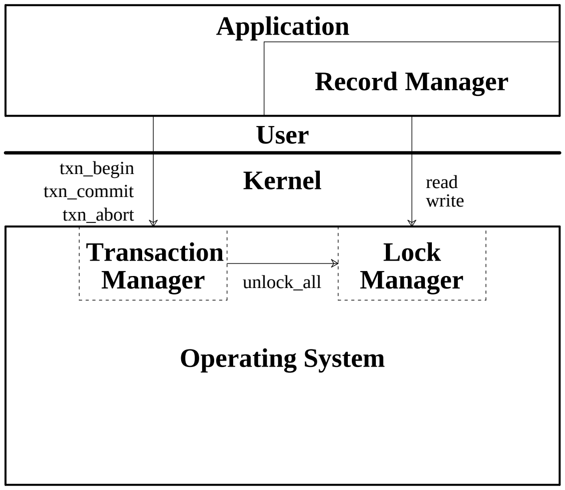

Figure 3: LFS Embedded Architecture.

The **lock table** maintains a hash table of currently locked objects which are identified by file and block number. Locks are chained both by object and transaction, facilitating rapid traversal during transaction commit and abort.

The **transaction state** is a per-transaction structure which describes the currently running transaction. It contains the status of the transaction (*idle*, *running*, *aborting*, *committing*), a pointer to the chain of locks currently held, a transaction identifier, and links to other transaction states.

The **inode** is used both in memory and on disk and describes the physical layout and other characteristics of a file. It is the physical representation of the index structure described in Section 2. The information contained in the inode includes such things as access times, modification times, file size, ownership, and protection. It is extended to include information which indicates if the file is transaction-protected. The in-memory representation of the inode additionally includes lists of buffers and links to other inodes. It is extended to contain a list of transaction-protected buffers, in addition to its normal buffer lists.

The **file system state** is an in-memory data structure which describes the file system and keeps track of its current state (e.g. read-only, current segment). It is extended to contain a pointer to the transaction lock table so that all transaction locks for the file system are accessible from a single point (as opposed to from each process with an active transaction).

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 id, and a list of open files for the process. It is extended to include a pointer to the transaction state.

4.2. Modifications to the Buffer Cache

The *read* and *write* system 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 satisfied (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.

4.3. The Kernel Transaction Module

Where the user-level model provides a subroutine interface, the embedded model provides a system call interface for transaction begin, commit, and abort processing. At *txn_begin*, a transaction structure is either created or initialized (depending on whether the process in question had previously ever invoked a transaction). 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.

At *txn_commit*, the same chain is traversed, but the dirty buffers are moved from the inode's transaction list to its dirty list. Once all the dirty buffers have been moved, the kernel flushes them to disk and releases locks when the writes have completed. In the case where only part of a page is modified, the entire page still gets written to disk at commit. This compares rather dismally with logging schemes where only the updated bytes need be written. While we might expect the increased amount of data flushed at commit to result in a heavy performance penalty, both the simulation results in [14] and the implementation do not indicate this. 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 are not noticeable in the resulting performance. Furthermore, forcing dirty blocks at commit obviates the need to write these blocks later when their buffers need to be reclaimed.

4.4. Group Commit

As database systems use group commit [3] to amortize the cost of flushing log records at commit, LFS can use group commit to amortize the cost of flushing its dirty blocks. Rather than flushing a transaction's blocks immediately upon issuing a *txn_commit*, the process sleeps until a timeout interval has elapsed or until sufficiently more transactions have committed to justify the write (create a larger segment).

4.5. Implementation Restrictions

It is important to notice that there are several deficiencies with the described implementation, most notably:

(1) All dirty buffers must be held in memory until commit.

(2) Locking is strictly two-phase and is performed at the granularity of a page.

(3) Transactions may not span processes.

(4) Processes have only one transaction active at any point in time.

(5) It is not obvious how to do media recovery.

For the benchmark described in this paper (a modified TPCB), requiring that buffers stay in main memory until commit does not pose a problem, because transactions are small and short-lived. With regard to page level locking, while the tests in Section 5 are not highly contentious, the simulation study in [14] indicated that locking at granularities smaller than a page is required for environments that are. Enhancements to the implementation described here that address these deficiencies are described in detail in [16].

5. Performance

The measurements reported here are from a DECstation 5000/200[2] running the Sprite Operating System [9]. The system had 32 megabytes of memory and a 300 megabyte RZ55 SCSI disk drive. The database utilized approximately 50% of the local 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, non-transaction, and sequential read performance. The transaction benchmark compares the LFS embedded system with the conventional, user-level

[2] DEC is a trademark of Digital Equipment Corporation

systems 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's write-optimized policy on sequential read performance.

5.1. Transaction Performance

To evaluate transaction performance, we used a modified version of the industry-standard TPCB transaction processing benchmark [18]. The test database was configured according to the TPCB scaling rules for a 10 transaction per second (TPS) system with 1,000,000 account records, 100 teller records, and 10 branch records. The TPCB benchmark simulates a withdrawal performed by hypothetical tellers 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 are updated to reflect the withdrawal and a history record, containing the account number, branch number, teller number, and the amount of the withdrawal, is written. 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-size record file (records are accessible sequentially or by record number).

This 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 1), providing a worst-case analysis. The configuration measured is so disk-bound that increasing the multi-programming level increases throughput only marginally. See [15] for a detailed discussion of the performance in a multi-user environment.

The interesting comparisons in this test are:

- User-level transaction manager on a log-structured file system and on a traditional file system.
- User-level transaction manager on LFS vs. kernel level transaction manager in LFS.

Figure 4 shows the results of this test. As expected, the LFS system outperformed the conventional file system, but by a disappointing 10% (12.3 TPS compared with 13.6 TPS). 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 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.

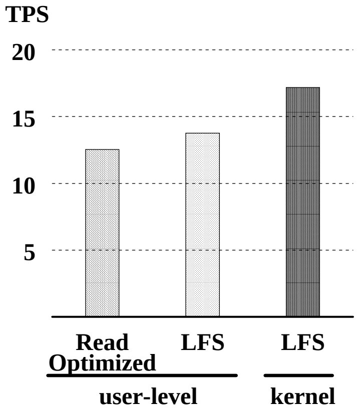

Figure 4: Transaction Performance Summary. The left and middle bars compare performance of a user-level transaction manager on the original Sprite file system (read-optimized) and on LFS. The middle and right bars compare the performance of the user-level transaction manager on LFS to the LFS embedded transaction manager.

The 10% performance improvement observed in this experiment is disappointing when compared to the disk bound simulation in [14] which predicted a 27% performance improvement. The difference between the implementation and the simulation can be explained by two factors. First, the simulation study did not account for the overhead of the cleaner. When the cleaner runs, it locks out all accesses to the particular files being cleaned. In this benchmark, there are only four files being accessed and these same files are the ones being cleaned. Therefore, when the cleaner locks these files, no regular processing can take place. As a result, periods of very high transaction throughput are interrupted by periods of no transaction throughput, and the resulting average is disappointing.

The second reason for the difference between the simulation and the implementation is that the simulation ignores much of the system overhead, focusing only 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 actual 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 of [14], 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 DECstation test platform does not have a hardware test-and-set instruction. As a result, the synchronization of the user-level model, used system calls to obtain and release semaphores, thus doubling the synchronization overhead of the kernel implementation which required a single system call. This synchronization overhead exactly accounts for the difference between the user and kernel implementations and is discussed in detail in [15]. Techniques described in [1] show how to implement user synchronization quickly on systems without hardware test-and-set. Such techniques would eliminate the performance gap shown in Figure 4.

5.2. Non-Transaction Performance

This test was designed to run programs that do not use the embedded transaction manager to determine if its presence in the kernel affects the performance of applications that do not use it. The non-transaction test consists of 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 [6], 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 compilations. 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).

All three benchmarks were run on an unmodified operating system and on one with embedded transaction support. Since the transaction code is isolated from the rest of the system, no difference in performance is expected. The results are summarized in Figure 5 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 accessing buffers to determine that transaction locks are unnecessary.

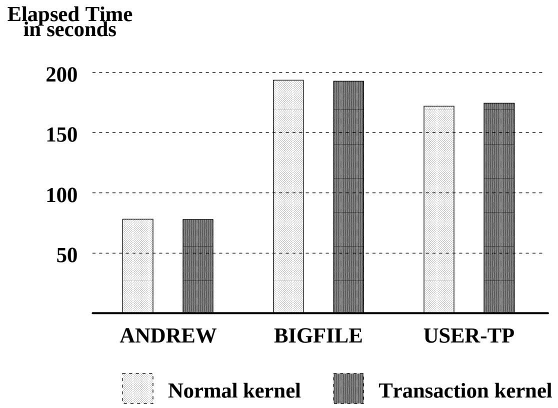

Figure 5: Impact of Kernel Transaction Implementation on Non-transaction Performance. None of the three benchmarks used the kernel transaction support. As is shown by the similarity in elapsed times for all benchmarks, the embedded support did not decrease the overall system performance.

5.3. 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 were executed, and then the file was read in key order. The account file is approximately 160 megabytes or 40,000 4-kilobyte pages, and the 100,000 transactions should have touched a large fraction of these pages, leaving the database randomly strewn about the disk.

Figure 6 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 50% better performance than LFS during the sequential test. (This test does not correspond exactly to reading the raw file sequentially since the file is read in key order.)

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. To quantify this tradeoff, we can

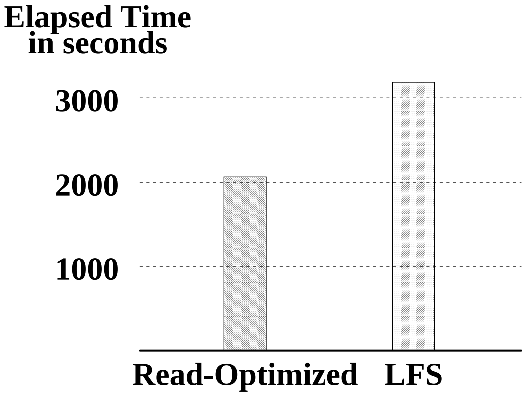

Figure 6: Sequential Performance after Random I/O. These tests were run after both systems had performed 100,000 TPCB transactions. The file system which favors sequential layout (read-optimized) was approximately 50% faster than LFS.

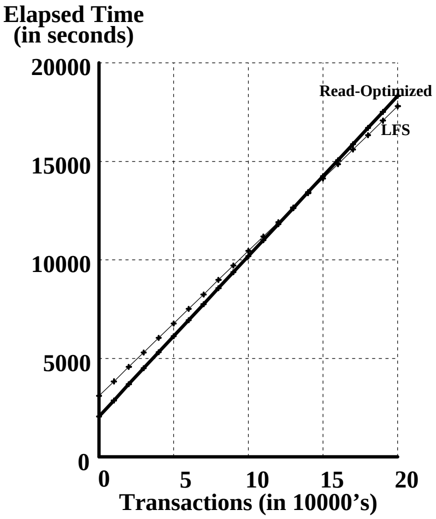

Figure 7: Total Elapsed Time for both Transaction Processing and Sequential Scan. Applications that require sequential processing after some period of transaction processing will observe better overall performance from the read-optimized system when the number of transactions executed is less than 134,300 and from LFS when more than that number are executed.

calculate how many transactions must be executed between sequential reads so that the total elapsed time for both file systems is the same.

Figure 7 shows the total elapsed time for both the transaction run and the sequential run 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, a 13.6 TPS 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 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, namely the cleaner, it seems obvious that this mechanism should be used to coalesce files which become fragmented.

5.4. Modifications to LFS

The issues raised in Sections 5.1 and 5.3 indicated some serious deficiencies in the current LFS design. As a result, a new version of LFS is discussed and analyzed in [17]. For the purposes of the specific issues highlighted here, the major design change is moving the cleaner into user-space.

Moving the cleaner into user-space accomplishes two goals. First, it prevents the cleaner from locking out other applications while the cleaner is running. Synchronization between the cleaner and the kernel occurs during a system call where cleaned blocks are checked against recently modified blocks. This should alleviate the disturbance in performance observed during the TPCB benchmark and improve LFS's overall performance.

Secondly, moving the cleaner into user-space makes it simple to experiment with different cleaning policies and implement multiple cleaners with different policies. So far, we have considered two cleaners. The first would run during idle periods and select segments to clean based on coalescing and clustering of files. The second would operate during periods of heavy activity using the same algorithm that was used for these experiments. Our belief is that removing the cleaner from the operating system and applying a variety of cleaning policies will enable us to close the gap in sequential read performance and to improve the random write performance in the transaction workload.

6. Conclusions

A log-structured file system shows potential for improving the performance of transaction processing workloads. Currently, LFS provides a 10% performance improvement over a conventional file system on a TPCB-style workload. While there are still several semantic issues to be resolved, embedding transactions in such a system also looks viable, with an embedded implementation performing comparably to the user-level system. Such an implementation enables applications to easily incorporate transaction-protection without rewriting existing applications. 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. While sequential read performance after random write performance still poses a problem for LFS, new designs offer promising solutions.

7. References

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] Marshall Kirk McKusick, William Joy, Sam Leffler, and R. S. Fabry, ‘‘A Fast File System for UNIX’’, *ACM Transactions on Computer Systems*, 2(3); August 1984, 181-197.

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

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

[11] 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.

[12] 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.

[13] Seltzer, M., Chen, P., Ousterhout, J., ‘‘Disk Scheduling Revisited,’’ *Proceedings of the 1990 Winter Usenix*, Washington, D.C., January 1990.

[14] Seltzer, M., Stonebraker, M., ‘‘Transaction Support in Read Optimized and Write Optimized File Systems’’, *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 1990.

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

[16] Seltzer, M., ‘‘File System Performance and Transaction Support,’’ PhD Thesis, University of California, Berkeley, December 1992.

[17] Seltzer, M., Bostic, K., McKusick, M., Staelin, C., ‘‘An Implementation of a Log-Structured File System for UNIX,’’ Proceedings of the 1993 Winter Usenix, San Diego, CA January 1993.

[18] Transaction Processing Performance Council, ‘‘TPC Benchmark B’’, Standard Specification, Waterside Associates, Fremont, CA., 1990.