The Logical Disk: A New Approach to Improving File Systems

Wiebren de Jonge

Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam

M. Frans Kaashoek and Wilson C. Hsieh

Laboratory for Computer Science, MIT, Cambridge, MA

Abstract

The Logical Disk (LD) defines a new interface to disk storage that separates file management and disk management by using logical block numbers and block lists. The LD interface is designed to support multiple file systems and to allow multiple implementations, both of which are important given the increasing use of kernels that support multiple operating system personalities.

A log-structured implementation of LD (LLD) demonstrates that LD can be implemented efficiently. LLD adds about 5% to 10% to the purchase cost of a disk for the main memory it requires. Combining LLD with an existing file system results in a log-structured file system that exhibits the same performance characteristics as the Sprite log-structured file system.

Keywords: Disk storage management, file system organization, file system performance, high write performance, logical disk, log-structured file system, UNIX, MINIX.

1. Introduction

We introduce a new approach for improving the structure and the performance of file systems that we call *Logical Disk (LD)*. LD provides an abstract interface to disk storage that uses *logical block numbers*, *lists of blocks*, *atomic recovery units*, and *multiple block sizes*. This interface separates the responsibilities of file and disk management, as illustrated in Figure 1: the file system is responsible for file management and LD is responsible for disk management.

Separating these two concerns has three advantages. First, it leads to a file system structure that makes file systems easier to develop, maintain, and modify. Second, it makes file systems more flexible. Different implementations of LD can be tailored for different access patterns and different disks can run under different implementations of LD. Similarly, different file systems can all share a particular LD implementation. Third, it allows for efficient solutions to the I/O bottleneck [Ousterhout and Douglis 1989; Ousterhout 1990]. LD can transparently reorganize the layout of blocks on the disk to reduce access time, similar to other systems that use logical-block numbers [Vongsathorn and Carson 1990; Solworth and Orji 1991; English and Stepanov 1992; Akyürek and Salem 1993].

Until recently these benefits have not been important, because operating systems, file systems, and disk drivers have all been tightly coupled in monolithic kernels. However, with modern kernel designs in which an operating system supports multiple file systems (e.g., Mach [Golub 1990] and Windows/NT [Custer 1993]), the benefits of separating file and disk management using LD are important.

To demonstrate these benefits we built a prototype implementation of LD. This implementation, *LLD*, is log-structured and is based on the ideas used in Sprite LFS [Rosenblum and Ousterhout 1992]. We combined LLD with MINIX [Tanenbaum 1987], a POSIX-compliant [IEEE 1990] file system for PCs. The resulting file system, *MINIX LLD*, required very few modifications to MINIX and exhibits the same performance characteristics as Sprite LFS; we expect MINIX LLD to outperform Sprite LFS, as it writes fewer blocks. However, it uses more main memory than Sprite LFS.

The contributions of this paper are threefold. First, we show that the LD allows a clean separation of file and disk management without loss of efficiency. Second, we show that an existing file system can easily be turned into a log-structured one by using a log-structured implementation of LD. Third, we introduce new implementation techniques for log-structured storage managers: a new recovery strategy and a new strategy for writing partially filled segments.

The last author was supported by the NSF under grant CCR-8716884, by DARPA under Contract N00014-89-J-1988, by an equipment grant from DEC, and by grants from AT&T and IBM.

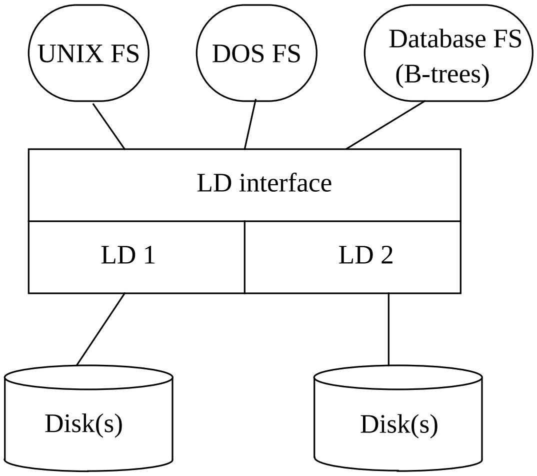

Figure 1. The LD interface separates file from disk management, which improves the structure of file systems. File systems become easier to develop, maintain, and modify, while performance can be improved transparently at the level of disk management. It also increases flexibility; multiple LD implementations, each tailored for different disk access patterns, can coexist, and multiple file systems can share a particular LD implementation.

The structure of the rest of the paper is as follows. Section 2 presents the primary abstractions provided by LD and a prototype version of the LD interface. Section 3 describes a log-structured LD implementation called LLD. Section 4 describes how we combined LLD with the MINIX file system to produce a log-structured file system, MINIX LLD. In Section 5, we compare related work (in particular, Sprite LFS, Loge [English and Stepanov 1992], and Mime [Chao et al. 1992]) and describe our ongoing and future work. In Section 6 we summarize our results and conclusions.

2. The Logical Disk

In this section we briefly describe abstractions supported by LD. We then describe the current LD interface.

2.1 Abstractions supported by LD

The four primary abstractions supported by LD are *logical block numbers*, *block lists*, *atomic recovery units*, and *multiple block sizes*.

File systems ask LD to write blocks at *logical* block addresses; LD chooses the *physical* locations on disk where the blocks will be written. To maintain the mapping between logical block numbers and physical disk-addresses, LD keeps a *block-number map*. Under this location-independent naming scheme, file systems address blocks by their logical addresses, which do not change, even if LD changes their physical locations.

The use of logical block numbers leads to a number of advantages. It enhances modularity: the file systems manages files and LD manages disks. LD can manage the disk transparently, as physical reordering of blocks can be accomplished by updating the block-number map.

Logical block numbers separate the names of blocks from their locations; as a result, cascading updates to file system data structures do not occur, which improves performance and simplifies implementations. In contrast to file systems that use LD, most file systems store physical disk addresses in their data structures. For example, if a data block is updated or moved in Sprite LFS, its physical address changes; blocks that reference such a block have to be updated as well. File systems that use LD store logical block numbers in their data structures, and do not see any cascading updates.

To allow file systems to express the logical relationships among blocks, LD supports the abstraction of *ordered lists* of logical blocks; LD can physically cluster the blocks according to the order in which they appear on a list. For example, a file system can put the blocks of a file on a list, and LD will try to place those blocks near each other physically; as a result, unnecessary disk seeks can be avoided when the file is read. The order of blocks on a list does not have to be the same as their order in a file; for example, it could correspond to the order in which they will be read. With lists the file system does not need to maintain partitions and cylinder groups [McKusick et al. 1984]. Finally, lists can be used to perform read-ahead effectively.

File systems can also specify the logical relationship between lists: LD keeps a single ordered list of lists, in which the file system inserts new lists. LD tries to physically place a list close to its neighbors in the list of lists.

Another abstraction supported by LD is the *atomic recovery unit*. During recovery all LD commands belonging to the same atomic recovery unit are treated as a single indivisible operation; LD will always recover to either a state that existed before or a state that existed after performing all operations of an atomic recovery unit. A file system can use atomic recovery units, for example, to treat the creation of a file and the update of its directory as a single operation. This eliminates the need for consistency checks such as those performed by *fsck*. In addition to helping file systems to keep their own state consistent, this facility can be used by file systems for supporting higher-level consistency mechanisms, such as application-level transactions.

LD also supports *multiple block sizes*, which allows different file systems to share an LD implementation. It can also be useful for a single file system; for example, multiple block sizes can be used to reduce internal fragmentation and to store i-nodes efficiently. A disadvantage of using small blocks is that the block-number map will require more memory.

2.2 A prototype LD interface

Table 1 lists the key primitives of the current LD interface. This interface does not fully support all of the abstractions and facilities we will eventually have in LD; we present the current interface to clarify our ideas. The interface is designed to support enough mechanisms to allow file systems to make a variety of policy decisions. The interface provides calls dealing with blocks, block lists, and some other auxiliary functions.

Of the calls dealing with blocks the most interesting are *NewBlock* and *DeleteBlock*. *NewBlock* returns a new logical block number, and takes two parameters. The first parameter specifies the list in which the new block should be inserted, and the second specifies where in the specified list it should be inserted. Since lists form the basis for physical clustering, these parameters can be viewed as hints about where to put the block on disk.

The last parameter to *DeleteBlock*, *PredBidHint*, is a hint for the predecessor of the block that is being deleted. For example, an LD could implement block lists using singly-linked lists. If the hint is correct, the block can be removed with one pointer update from the list. If no hint is given or if the hint turns out to be incorrect, LD searches for the predecessor from the beginning of the list.

NewList and *DeleteList* deal with block lists. *NewList* allocates a new list and returns its list identifier. To allow for interlist clustering, the file system can specify where in the list of lists the new list should be inserted. *NewList* also takes a hint parameter, which indicates whether blocks in the list should be clustered or compressed, and whether interlist clustering should be applied to this list.

DeleteList deletes a previously allocated list and its blocks. Like *DeleteBlock*, *DeleteList* takes a hint to find the predecessor efficiently.

Block lists are a powerful abstraction that allows LD to perform *interfile* and *intrafile clustering*. For example, using one or more lists file systems can easily achieve all or most of the benefits of using cylinder groups [McKusick 1984]. One simple way of using lists is to have one list for all blocks, where the blocks appear in the same order as in their files. When a new file is created, the file system inserts its first block immediately after the last block of some other file (e.g., one in the same directory).

Another option is to have a single list per file. When the file system calls *NewList* to create a list, it can specify the list identifier of another file as a predecessor in the list of lists; LD can place the lists close together. Subsequent calls to append new data blocks to a file use the new list identifier and can specify the block number of the file's last block, so that all the file's data blocks will be clustered.

Many other options are possible, such as using a list for a group of files. The use of lists offers more flexibility than using cylinder groups; with cylinder groups the physical size of each group is fixed and the physical boundaries are determined statically.

Command	Description
Read(Bid, Buf, Cnt)	Read logical block *Bid*.
Write(Bid, Buf, Cnt)	Write logical block *Bid*.
NewBlock(Lid, PredBid)	Allocate a logical block number *Bid* and put the block on the list *Lid* after the logical block *PredBid*. If the call succeeds, return logical block number *Bid*.
DeleteBlock(Bid, Lid, PredBidHint)	Remove a block and free its block number. Block number *PredBidHint* is a hint to find the predecessor efficiently; if it is incorrect, use list identifier *Lid* to search the list for the predecessor.
NewList(PredLid, Hints)	Allocate a list. If the call succeeds, it returns a list identifier *Lid*. The list is inserted in the list of lists after the list *PredLid*. The file system can pass hints to indicate, for example, whether the blocks in this list should be compressed and/or clustered, and whether the list itself should be clustered near its predecessor.
DeleteList(Lid, PredLidHint)	Free the list named by *Lid*. List identifier *PredLidHint* is a hint to find the predecessor efficiently.
BeginARU()	All commands up to the next *EndARU* will belong to the same explicit atomic recovery unit.
EndARU()	End explicit atomic recovery unit.
Flush(FailureSet)	After a successful return of *Flush* the results of all previous commands are guaranteed to survive the given kinds of failures.

Table 1. The key primitives in the prototype LD interface, which supports logical blocks, lists of blocks, atomic recovery units, and multiple block sizes. *Bid* stands for block identifier. *Lid* stands for list identifier. *PredBid* and *PredLid* use a special value to specify insertion at the beginning of the list and list of lists, respectively.

BeginARU and *EndARU* mark the beginning and the end of an explicit atomic recovery unit. The LD operations in an atomic recovery unit are treated as an indivisible operation during recovery; i.e., LD guarantees that all or none of these operations are persistent. The interface does not yet support concurrent ARUs.

After a file system has called *Flush* LD guarantees that all preceding operations are persistent.

Our prototype interface has a number of additional primitives that are not listed in the Table 1. There are two primitives for reserving physical disk space for logical blocks and for cancelling such reservations. These primitives address the problem that most UNIX file systems cannot handle *write* calls that cannot succeed due to lack of disk space. Three other primitives allow the file system to move sublists of blocks in and between lists, to move whole lists to another logical position in the list of lists, and to flush a list. The first two primitives allow the file system to easily express changes in requested clustering; the last primitive allows an easy implementation of *fsync*. Finally, there are primitives to initialize and shut down LD.

3. A Log-structured LD Implementation

Different implementations of the LD interface are possible; we describe an implementation that is based on the ideas in Sprite LFS. This implementation, which we will refer to as *LLD*, is based on the assumption that most reads are absorbed by the file system cache and that therefore disk traffic is dominated by writes. This section shows that a log-structured implementation of LD is feasible and also introduces new implementation techniques for log-structured file systems.

LLD, like LFS, writes dirty file system blocks to disk in long contiguous writes. Figure 2 illustrates the primary LLD data structures. LLD divides the disk into large, fixed-size *segments*; the segment being filled is maintained in main memory and is written in a single disk operation. Each segment contains a *segment summary* that is used as a log for LLD's metadata. For example, it is used to record information about each block in the segment. During idle periods the *reorganizer* will try to improve the layout of blocks and lists on disk and to clean segments, so that empty segments remain available. If LLD runs out of empty segments while busy, it will call the *segment cleaner*, which produces an empty segment as quickly as possible. To determine which segments to clean, LLD maintains in main memory a *segment usage table* that records the number of live bytes in each segment. The lists that are maintained in the block-number map and the list table are implemented as singly-linked lists.

3.1 Normal operation

When the file system calls *NewBlock*, LLD removes a free logical block number from its list of free numbers and inserts this logical block number into the specified list. The

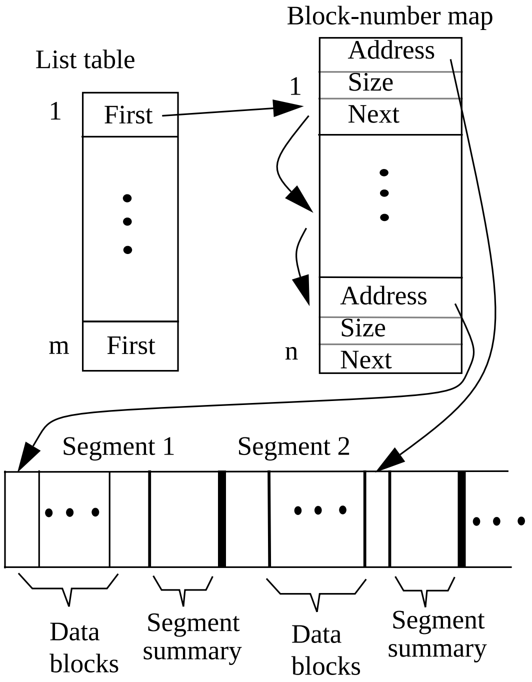

Figure 2. Data structures in LLD. The block-number map and the list table are maintained in main memory. For each logical block the block-number map stores its physical address, its successor in its list, its length, and whether it is compressed. The list table stores the number of the first logical block on each list. Segments are stored on disk. A segment is a fixed-size structure that contains data blocks and a segment summary. Segments store a variable number of blocks, which are variable-sized to support compression. The segment that is currently being filled is maintained in main memory, and is written to disk in a single operation. The segment summaries are used for logging updates to LD's metadata. For example, a summary stores its own size, the number of logical blocks in the segment, and for each physical block in the segment its logical block number, timestamp, length, and a flag indicating whether its contents are compressed. In addition, the segment summary logs list modifications.

physical address of the block and its size are registered when the block is actually written.

When the file system gives LLD a block to write, LLD copies the block to the segment in main memory and records its logical block number and a timestamp in the segment summary in main memory. It then updates its block-number map by storing the new physical address of the block. When the current segment becomes full, LLD writes it out in a single disk operation.

When the file system needs to read a block that is not in its cache, it calls LLD with the logical block number of the block to be read. LLD uses the number as an index in the block-number map and finds the physical location of the block. It then reads the block stored at that location or finds it in the segment in main memory, and returns it to the file system.

For operations that modify a list, such as *NewBlock*, LLD creates *link tuples* that contain a timestamp, a block number, and the new value for the successor field. These tuples are inserted into the segment summary in main memory, and are written to disk when the segment is written to disk. Using the link tuples, LLD can reconstruct the lists during recovery.

All information in a segment summary, such as link tuples and logical block numbers, is not only timestamped, but is also tagged with a bit indicating whether or not this information ends an atomic recovery unit. After a *BeginARU* everything until the next *EndARU* is tagged as not ending an atomic recovery unit. This information is sufficient for LLD to guarantee the “all-or-nothing” semantics of atomic recovery units.

3.2 Partial segments

As in Sprite LFS, the idea in LLD is to collect many dirty blocks in the segment in main memory and to write the segment to disk in a single operation. Due to *Flush* calls *partial segments* (i.e., segments that are only partially filled) may have to be written. If this happens frequently, then cleaning also has to be performed more frequently. To reduce this extra cleaning LLD uses a simple but effective solution. If a segment is filled above a certain threshold — for example, 75% of its capacity — LLD treats it as a full segment and writes it to disk. If the segment is filled below the threshold, LLD writes the segment out, but keeps the segment in main memory as if it had not been written. When the segment is full (or *Flush* is called again after the threshold has been passed), the whole segment is written to disk and the partial segment written earlier is freed. The physical segment used for the partial write can then be reused without any cleaning overhead.

Using this approach the costs of a *Flush* are as follows: if the segment was filled above the threshold, then no more bandwidth than strictly necessary will be lost; on the other hand, if the segment was filled below the threshold, the overhead of the *Flush* will involve an additional seek and an additional write. The average cost of a *Flush* is highly dependent on the rate at which *Flush* is called, since that rate influences the average “fullness” of a segment. At high rates *Flush* calls will be costly. However, if *Flush* is called too frequently, log-structured file systems in general will not perform well, since large amounts of data do not get written in a single write. Fortunately, the need for doing *Flush*’s is lessened when making use of LD’s atomic recovery units.

The main advantages of this approach for writing partial segments are that it is simple to implement, that recycling of partial segments is inexpensive, and that segment summaries can be kept in fixed locations on the disk. This last property is vital for LLD’s approach to recovery. The disadvantage is that frequent *Flush*’s will cause blocks to be written multiple times, which does not occur in Sprite LFS.

3.3 Compression

To demonstrate the flexibility that the LD interface offers, we have implemented compression, as has been done for Sprite LFS [Burrows et al. 1992]. We use an algorithm due to Wheeler for its simplicity and performance; the algorithm is described by Burrows et al. Using LLD, a file system can transparently use compression to make more effective use of disk space. LLD compresses user data and file system data structures (e.g., i-nodes), but not its own data structures. To support compression LLD internally uses variable-sized blocks.

If a file system specifies that the blocks in a list should be compressed, LLD compresses those blocks into the segment in main memory instead of copying them. It also records in the block-number map and the segment summary that the block is compressed and what its length is after compression. When reading a block, LLD checks in the block-number map whether the block is compressed. If so, LLD decompresses it before returning it to the file system. LLD currently compresses all blocks of a list during writing; it may be a better strategy to only compress cold (not recently referenced) blocks during cleaning. Under such a strategy writes and reads of recently written data can be performed at the maximum disk bandwidth instead of being restricted by the bandwidth of the compression algorithm. However, as processor speeds increase the compression bandwidth will increase and will not be a bottleneck.

3.4 Memory and disk space requirements

Data structure	**LLD using single list**	**LLD using compression and one list per 8 Kbyte file**
Block-number map	1.5 Mbyte	3.8 Mbyte
List table	4 byte	0.8 Mbyte
Segment usage table	6 Kbyte	6 Kbyte
Total	1.5 Mbyte	4.6 Mbyte

Table 2. Main memory used by LLD per Gbyte of physical disk space for different configurations, assuming an average block-size of 4 Kbyte and a compression ratio of 60%. In case of compression the figures are per 1.7 Gbyte of actual storage space.

The data structures of our LLD implementation require a substantial amount of main memory (see Table 2). Without support for compression each logical block uses three bytes for its physical block address and three bytes for its successor. With a 1-Gbyte disk and an average block-size of 4 Kbyte, the block-number map requires 1.5 Mbyte of memory. To support compression at most two bytes are needed to store the length, an additional byte is needed for the physical address, and 67% more blocks will fit (assuming the compression ratio is 60%); in this case the block-number map requires 3.8 Mbyte of main memory. Storing the list table takes four bytes per file. The costs of the list table are negligible if one list is used for a whole file system or if each list is used for a sufficiently large group of files; if one list is used per file, and if one assumes an average file size of only 8 Kbyte, then the list table would take 0.8 Mbyte when using compression. This computation is based on our current implementation, which does not keep the list of lists. The segment usage table takes three bytes per segment, i.e., 6 Kbyte (assuming 512-Kbyte segments). Thus, in total the memory requirements for storing every data structure completely in main memory are at least 1.5 Mbyte and at most 4.6 Mbyte per Gigabyte of physical disk space. Given today’s prices LLD adds from 3% to 31% to the price of a disk (see Table 3); however, with compression the file system gets 1.7 Gbyte of actual storage space.

Price of a Mbyte RAM	**Price of a Gbyte disk space**	
	$750	**$1500**
$30	6% or 18%	3% or 9%
$50	10% or 31%	5% or 15%

Table 3. The percentage cost that LLD adds to the cost of disks for different prices of main memory and disk space, assuming the best case and the worst case for LLD: 1.5-Mbyte per Gbyte of disk space (no compression and a single list for all files) and 4.6-Mbyte RAM per Gbyte of disk space (compression and a list per file).

Less memory-intensive solutions are possible. For example, LLD could store data structures on disk and could cache in main memory only what is actively used. The list table seems to be a good candidate for this, as probably only a small percentage of the lists will be actively used at any moment in time. Ruemmler and Wilkes analyzed UNIX block access patterns and observed that 1% of the blocks receive 90% of the writes [Ruemmler and Wilkes 1993]. Although they did not analyze the variation in the set of blocks that are written, this suggests that caching the block-number map could be effective as well.

In comparing LLD to other file systems, one has to take into account that the money spent on main memory for LLD (which results in better structure, flexibility, and performance) could instead be used in other file systems to improve performance. For example, one could buy more memory to cache data and metadata or one could buy non-volatile RAM to absorb disk writes. Although spending money in this way will improve performance, it will not improve structure and flexibility.

We now examine the disk space requirements. Without compression, each physical block requires seven bytes in the segment summary, three for its logical number and four for its timestamp. In addition, link tuples are stored in the segment summary, which is done with 12 bytes per tuple. With a 0.5-Mbyte segment, 4-Kbyte blocks, and no link tuples, the segment summary would be 889 bytes. The number of link tuples varies per segment, since the number of list operations that are performed while the segment is being filled is not fixed, but a segment summary of one 4-Kbyte block leaves room for 267 tuples. With compression in LLD, each physical block requires three additional bytes, i.e., 10 bytes. Assuming Wheeler’s algorithm achieves a compression ratio of about 60% [Burrows et al. 1992], a segment contains on average 211 compressed blocks (i.e. 2,110 bytes for the block entries), leaving room in a 4-Kbyte segment summary block for 165 link tuples.

3.5 Cleaning and clustering

Rosenblum and Ousterhout describe a number of policies for selecting and cleaning segments using the segment usage table [Rosenblum and Ousterhout 1992]; all of these can be used for LLD as well. Using the lists, LLD can physically cluster related blocks.

We have not yet implemented the disk reorganizer, which should clean segments and improve the layout of blocks and lists on the disk during idle periods. Our current LLD implementation uses a segment cleaner that cleans a number of segments and uses a simplistic clustering strategy: when it copies blocks from a segment that is being cleaned, it uses the list information to reorder the blocks to improve sequential read performance. LLD also removes old logging information, such as old link tuples and old *EndARU* tuples, from the segment summaries during cleaning.

3.6 Recovery

LLD can be shut down explicitly in a clean state or implicitly due to some failure. If LLD is shut down explicitly, it writes its data structures, a timestamp, and a marker that the state stored is valid in a special region on disk. During recovery LLD determines whether it was shut down in a clean state or whether a failure has happened.

In the case of explicit shut down, LLD reads its data structures from the special area on disk, invalidates the marker, and starts immediately.

After a failure LLD reads all of the segment summaries in a single sweep over the disk and rebuilds its data structures from the information stored therein. For each disk block registered in a segment summary, LLD uses its timestamp to determine whether it is the most recent version of a logical block, so that the segment usage table and the physical address part of the block-number map can be reconstructed. Similarly, LLD uses the link tuples in the segment summaries to restore the most recent links in the list table and the block-number map.

The support of atomic recovery units makes recovery after failure slightly more complicated. While reading the segment summaries during recovery LLD keeps the timestamp of the most recently committed operation. When LLD encounters an incomplete atomic recovery unit, its effects are deferred if the atomic recovery unit is more recent than the most recently committed operation; the atomic recovery unit is queued until LLD encounters its *EndARU* or a more recently committed operation. If there is an incomplete atomic recovery unit after recovery, LLD cleans its operations from their segments.

LLD's approach to recovery has the advantage that no checkpoints are needed during normal operation. Furthermore, modified blocks of its own data structures do not need to be written out with blocks containing user data. Therefore, the effective transfer rate of user data is higher than for Sprite LFS. Our approach to recovery also has the potential of being fast. With a proper layout of the segments over the cylinders, only one revolution per cylinder would be needed for reading all segment summaries.

Although LLD currently uses the "one-sweep" approach to recovery, it is in no way restricted to do so. LLD could as easily use another recovery strategy.

4. Experimental Evaluation

To verify our claims we combined LLD with the MINIX file system, and ran the same microbenchmarks used to measure Sprite LFS. We first describe MINIX LLD and then we present the performance results.

4.1 MINIX LLD

The MINIX file system was developed for PCs; we ran it as a user-level process on top of UNIX. It uses two bitmaps to keep track of free disk space: one for free i-nodes and one for free blocks. When it allocates a block for a file, it allocates it close to the previous allocated block for that file. MINIX keeps recently used data and i-node blocks in a buffer cache, which is flushed when an application calls *sync*.

We have implemented LLD as a user-level process and linked it to the MINIX file system. LLD writes and reads blocks from a disk partition using UNIX system calls.

We initially made four changes to MINIX so that it could use LLD (in addition, we fixed three performance bugs in MINIX after we started taking measurements). First, MINIX calls *NewBlock* to allocate a new block for a file; it also tells LLD to add the block to the list (initially MINIX LLD used a single list for all files). Second, when MINIX frees a block it notifies LLD that the block is free. Third, upon a *sync* MINIX tells LLD to flush the segment that is currently being filled; this ensures that after a user or the file system has called *sync*, all data are on the disk. Fourth, read-ahead in MINIX is disabled, since blocks that MINIX thinks are contiguous may not actually be so.

We later made three additional changes to MINIX so that it fully uses the LD interface. First, MINIX stores each file's blocks in a separate list, which allows for better clustering. Second, MINIX stores the list identifier in the i-node, so that it can remember the list identifier for each file. Third, MINIX no longer stores the block bitmap to keep track of free disk space.

We also experimented with two block sizes: one for regular data blocks and one for i-nodes. This change improves the write performance slightly as MINIX can ask LLD to write a single i-node instead of a complete i-node block, but it may decrease read performance, as each i-node is read separately from disk. It also increases the size of the block-number map; for 10,000 i-nodes LLD (with compression supported) needs at most an additional 120 Kbyte.

Although LLD supports atomic recovery units, MINIX does not make use of them yet. In total less than 100 of the 7000 lines of general file system code were modified. The end result was that the MINIX file system became simpler, as most of the disk management code (350 lines) could be deleted.

4.2 Performance of MINIX LLD

To measure MINIX LLD's performance we ran the same microbenchmarks as Rosenblum and Ousterhout [Rosenblum and Ousterhout 1990]. These benchmarks measure the performance of specific file operations and not overall system performance [Seltzer 1992]. However, these benchmarks are good enough to demonstrate that combining LLD with an existing file system achieves the same performance characteristics as a log-structured file system.

The first benchmark measures small file I/O: the cost of creating, reading, and deleting 10,000 1-Kbyte files and 1,000 10-Kbyte files in one directory. The second benchmark measures large file I/O; it measures writing and reading an 80-Mbyte file from a newly created file system in five stages: write an 80-Mbyte file sequentially; read the file sequentially; write 80-Mbyte randomly to the file; read 80-Mbyte randomly; and read the file sequentially again. We ran MINIX LLD in two configurations: one in which MINIX collects i-nodes in one disk block and one in which MINIX allocates a 64-byte block for each i-node. To evaluate the performance of MINIX LLD we also ran the benchmarks for the MINIX and SunOS 4.1.3 file systems.

The measurements were carried out on a 33-Mhz SPARC-10/20 workstation with 64-Mbyte main memory running SunOS 4.1.3 (note that SunOS 4.1.3 performs better than version 4.0.3, the version of SunOS to which Rosenblum and Ousterhout compared Sprite LFS); both MINIX file systems were measured on a disk partition of 400 Mbyte on a 2-Gbyte disk (HP C3010: SCSI-II, 5400 rpm, 11.5 msec average seek time). The SunOS file system ran inside the kernel, while MINIX and MINIX LLD ran as user-level processes that used the raw disk interface provided by SunOS.

MINIX LLD used 0.5-Mbyte segments and 4-Kbyte blocks, MINIX used 4-Kbyte blocks, and SunOS used 8-Kbyte blocks. To eliminate the effects of the differences in file caches, we flushed the file cache before each phase in the experiments by writing and *sync*ing a file several times larger than the file cache. Both MINIX and MINIX LLD used a static buffer cache of 6,144 Kbyte, while the SunOS buffer cache grew and shrank dynamically. A user-level process writing 0.5 Mbyte segments to the disk partition in a tight loop achieves a throughput of 2400 Kbyte/s on this configuration. We ran each experiment at least 10 times. The standard deviation for almost all of the experiments was below 1% of the mean; all deviations but one were below 4%.

Table 4 shows the results of running the benchmark for small file I/O in files per second. The numbers for MINIX and MINIX LLD include neither the overhead of the application itself, nor that of the pipes between the application and the file system so that we could measure the exact difference between MINIX and MINIX LLD.

Creation of files is faster in MINIX LLD than in MINIX because MINIX LLD collects many changes in a single write. File reading in MINIX LLD has the same speed as in MINIX because reads are performed sequentially in both file systems. File deletion in MINIX and MINIX LLD have similar performance for two reasons. First, MINIX only makes directory changes stable at *sync*'s; all the modified i-nodes can be written together. Second, MINIX performs a few more seeks, but MINIX LLD incurs overhead for maintaining list information.

The numbers in the table for SunOS are worse than for the two MINIX file systems. Creation and deletion are worse since SunOS performs these operations synchronously. Reads are probably worse due to unsuccessful read-ahead.

Table 5 shows the results for large file I/O in Kbyte per second. Like Sprite LFS, MINIX LLD shows excellent performance on all writes, as all file writes are turned into sequential disk writes; MINIX LLD uses 85% of the available bandwidth. MINIX, on the other hand, uses only 13% of the available bandwidth. MINIX's throughput is low because the disk must make an additional rotation between writing two consecutive 4-Kbyte blocks (a program that writes back-to-back 4-Kbyte blocks to the disk achieves a throughput of only 300 Kbyte per second); the same phenomenon was detected in the original Sprite file system [Rosenblum 1992]. If we improved MINIX's block allocation strategy as in FFS [McKusick et al. 1984], MINIX should achieve similar performance on sequential writes as MINIX LLD.

File System	10,000 1-Kbyte file (file/sec)			1,000 10-Kbyte file (file/sec)		
	C	**R**	**D**	**C**	**R**	**D**
MINIX LLD	288	227	806	134	97	680
MINIX	70	215	784	25	100	800
SunOS	20	132	40	16	74	28

Table 4. Performance results in file/sec for creating (**C**), reading (**R**), and deleting (**D**) 10,000 1-Kbyte and 1,000 10-Kbyte files in one directory. Higher numbers are better.

File System	Write Seq.	Read Seq.	Write Rand.	Read Rand.	Read Seq.
MINIX LLD	2030	1300	2000	401	445
MINIX	316	1460	324	220	1460
SunOS	2948	3724	639	439	3747

Table 5. Performance results in Kbyte/sec for writing and reading a 80-Mbyte file (in 8-Kbyte chunks) for MINIX LLD and MINIX; for SunOS a 300-Mbyte file was used to reduce the effects of its larger file cache. Higher numbers are better.

MINIX achieves higher throughput than MINIX LLD on sequential reads because it uses prefetching, which we disabled for MINIX LLD. The performance of random reads for MINIX LLD is better than for MINIX because MINIX's read-ahead strategy fails. The performance for the sequential reads after the random writes are better for MINIX than MINIX LLD, since MINIX updates blocks in place and thus overwriting does not change their order; therefore, MINIX performs fewer seeks when reading the blocks, and prefetching works well.

SunOS performs better on sequential writes and all reads than the MINIX file systems: it has more disk bandwidth available, since MINIX depends on SunOS to access a raw disk partition. For random writes the performance of SunOS is worse than MINIX LLD, because MINIX LLD turns random writes into sequential writes.

In addition to running the microbenchmarks, we measured the time for MINIX LLD to start after a failure. The combined time for LD and MINIX to recover was 12 seconds. This number measures the cost of reading 788 segment summary blocks (including the list information), building up the block-number map, and reading the superblock, root i-node, and initializing the MINIX file system data structures.

The cost for using lists depends on the frequency of block allocation and deallocation. To measure the costs of supporting lists, we also ran the benchmarks for a version of MINIX LLD that does not support lists. Different runs of the benchmark have shown that there is little overhead during reading or writing. There is only significant overhead during block allocation and deallocation; during the create and delete phases of the small file benchmarks the overhead for maintaining lists was approximately 15%.

We measured performance for a range of segment sizes. The differences in performance for 128-Kbyte, 256-Kbyte, and 512-Kbyte segments are within a few percent. Smaller segment sizes result in a loss of write performance. For 64-Kbyte segments we measured a reduction in write performance of 23%.

We measured a version of MINIX LLD that allocates each i-node as a small block. This version can improve write performance, as MINIX can make better use of the write bandwidth by writing a single i-node instead of a complete i-node block. However, this version performs the same for write operations and worse for read operations on the small-file benchmarks. Performance for creating and deleting files are similar, as the clustering that MINIX performs pays off for this benchmark (many dirty i-nodes share a single i-node block). Read performance on the small-file benchmarks is worse, since blocks are misaligned (the disk is a block device) and each i-node is read individually. This version of MINIX LLD exhibits the same performance on the large-file benchmark, since this benchmark operates on a single file (only one i-node is allocated).

We also measured the throughput of MINIX LLD with compression*;* the write throughput was 1600 Kbyte per second, and the read throughput was 800 Kbyte per second. The write throughput is within 21% of the throughput without compression; this is because one segment can be compressed while the previous segment is being written to disk. The read throughput is low because we cannot overlap reading and decompression. Compressing every block is likely to be a good choice in systems with sufficiently large file caches or hardware support for compression; in systems with small file caches, it would be better to only compress cold files during cleaning.

The measurements show that MINIX LLD indeed makes more effective use of the disk bandwidth than MINIX. With only four small modifications to MINIX, we combined it with LD and turned it into a log-structured file system. A main point of this paper is confirmed by the experiments: with LD an existing file system can easily be made more efficient.

5. Comparison and Discussion

In the previous section we discussed the measured performance of MINIX LLD; we did not do a head-to-head performance comparison of MINIX LLD and Sprite LFS. This section begins with a discussion of Sprite LFS and MINIX LLD. We then compare LD and LLD with Loge and Mime; we describe other related work and we conclude with a discussion of ongoing and future work.

5.1 Comparison with Sprite LFS

Table 6 compares Sprite LFS and MINIX LLD on a number of issues, each of which we discuss in turn.

To create a file with no data in an existing directory, or to delete an empty file, Sprite LFS writes the updated data block of the directory, two dirty i-nodes, and in general two updated blocks of the i-node map (Sprite LFS stores the physical location of each i-node in the i-node map and writes modified blocks of the i-node map to disk during a checkpoint). The i-nodes must be written out because the disk addresses and the modification times stored in them have changed. Since in Sprite LFS dirty i-nodes are collected together in special blocks [Rosenblum 1992], the cost of writing a dirty i-node is much smaller than the cost of writing one block and will be denoted by ε. Sprite LFS writes the modified i-node map blocks to disk only on a checkpoint and thus multiple i-nodes and multiple operations per i-node may share the same i-node map block; the cost per operation for one of these blocks is denoted by δ, where δ is a value between 0 and 1. The actual cost of file creation is therefore $1{+}2\delta{+}2\varepsilon$ blocks.

For the same operation MINIX LLD writes the updated directory block and two dirty i-nodes. MINIX LLD avoids cascading updates, but still writes the i-nodes in order to make their modification times recoverable, as required by the POSIX standard. The cost of writing an i-node is the same as in Sprite LFS (if MINIX LLD uses small blocks for i-nodes); the total cost is thus $1{+}2\varepsilon$ blocks.

The overhead of cascading updates appears when Sprite LFS overwrites an existing data block in a file. In this case the additional overhead is the i-node map block and potentially the indirect block and the double-indirect block. For MINIX LLD there are no cascading updates, and none of these blocks need to be rewritten.

In Sprite LFS appending a block to a file results in overhead for writing the i-node and the i-node map block. Depending on the file size indirect and double-indirect blocks may also have to be written, as inserting the physical address of a new block into an indirect block will cascade to any corresponding double-indirect block. In MINIX LLD the block, the i-node, and an indirect block are written when appending a block to a large file; only when a new indirect block is needed does a double-indirect block has to be written.

Issue	Comparison
Reading a block	Performance is equal for Sprite LFS and MINIX LLD.
Creating or deleting a file	Sprite LFS writes 1+2δ+2ε blocks; MINIX LLD writes 1+2ε blocks.
Overwriting a block	Sprite LFS writes 1+δ+ε, 2+δ+ε or 3+δ+ε blocks; MINIX LLD always writes 1+ε blocks.
Appending a block	Sprite LFS writes 1+δ+ε, 2+δ+ε or 3+δ+ε blocks; MINIX LLD usually writes 1+ε or 2+ε blocks. MINIX LLD only writes 3+ε blocks in the rare case that a new indirect block is needed for appending a double-indirect data block.
Simplicity	Sprite LFS requires significant changes to the general file system code. MINIX LLD requires only minor modifications.
Memory requirements	MINIX LLD uses 4 to 6 Mbyte more main memory for its data structures than Sprite LFS.
Cleaning and clustering	MINIX LLD requires less segment cleaning than Sprite LFS, since it fills segments with more user data. Reorganization is also easier with MINIX LLD.
Recovery	Sprite LFS may recover somewhat faster, but requires checkpoints during normal operation.

Table 6. Comparison between Sprite LFS and MINIX LLD. The cost in Sprite LFS for overwriting a block will be lower when many data blocks belonging to the same file are written at once. ε denotes a small value. δ denotes a value between 0 and 1.

A more precise comparison would take into account that a segment can be shared by many data blocks belonging to the same file, and that both implementations could be improved. For example, when a large file is completely overwritten, the overhead in Sprite LFS will be only a few blocks per segment (one special block with the updated i-node, one or two indirect blocks and possibly a double-indirect block). If the segment size is 100 blocks or more, the performance advantage of MINIX LLD for this last example is very small. As another example, the implementation of Sprite LFS could be improved by writing out indirect blocks only on checkpoints. This reduces the overhead for cascading updates, but it has the disadvantage that checkpoints become more time consuming and that data blocks and their metablocks are no longer colocated.

An advantage of LLD is that it requires almost no modifications to the general file system code, whereas Sprite LFS requires significant changes to the general file system code. This benefit is due to the fact that LD separates disk and file management, which results in a more modular system. Using LLD a file system can easily be turned into a log-structured one.

A disadvantage of MINIX LLD is that it may use more main memory than Sprite LFS, as the block-number map contains one entry per block, whereas the i-node map contains one entry per i-node. For a 4-Gbyte disk and 4-Kbyte blocks and assuming that each user file is on average 8 Kbyte, a simple implementation of LD that does not support compression uses 6 Mbyte of main memory to store the block-number map and 2 Mbyte to store the list table (Section 2 computes the memory requirements for 1-Gbyte disk space). Assuming that each i-node map entry is 12 bytes, the i-node map in Sprite LFS for a 4-Gbyte disk with 4-Kbyte blocks requires 6 Mbyte of memory [Rosenblum 1992].

However, Sprite LFS stores the i-node map on disk and caches it on a block-by-block basis, so it need not be entirely memory resident. To make caching effective Sprite LFS allocates i-node numbers to maximize locality (e.g., files in the same directory have nearby i-node numbers). For an 8.1-Gbyte disk Sprite LFS caches on average only 3.6-Mbyte of the i-node map [Rosenblum 1992]. MINIX LLD currently keeps the entire block-number map and list table in main memory.

As a consequence, compared to Sprite LFS, MINIX LLD without compression needs 4 or 6 Mbyte of additional main memory depending on whether one list is used for all files or one list for each file. In both cases the use of LLD leads to a cleaner structure of the file system as well as higher performance. It is also easy to add compression to MINIX LLD at a price of 12 Mbyte of main memory, which gives the file system an additional 2.6 Gbyte storage space.

Segment cleaning and physical clustering are more efficient in MINIX LLD than in Sprite LFS. During cleaning and reorganization live data blocks will be moved, and their physical addresses will change. Unlike in MINIX LLD, in Sprite LFS the related i-nodes, indirect blocks, and double-indirect blocks always have to be updated and moved.

Sprite LFS periodically stores the physical locations of all of the blocks in the i-node map in a checkpoint region. Under this approach recovery is fast, since roll-forward only has to be done from the last checkpoint. MINIX LLD does not take checkpoints during operation; therefore, recovery after a failure may take more time, since MINIX LLD reads all of the segment summaries to reconstruct its block-number map. Assuming a 4-Gbyte disk and 4-Kbyte blocks, Sprite LFS reads up to about 750 blocks (3 Mbyte) for the i-node map and then reads all segments written since the last checkpoint, while MINIX LLD reads about 8,000 segment summaries. MINIX LLD however does not have to roll forward; as shown by the measurements in the previous section, MINIX LLD's recovery strategy is still quite fast.

5.2 Comparison with Loge and Mime

Loge [English and Stepanov 1992] improves the I/O performance of disks by having the disk controller reduce the time required for writing a stream of individual blocks. For its internal operation Loge typically reserves 3-5% of the physical blocks [Chao et al. 1992]. When a block is written to disk, the Loge disk controller writes to a free reserved physical block closest to the current position of the disk head. The number of free reserved physical blocks remains constant; each time a logical block is written to a physical one, its previous physical location becomes free and will then be reserved for future internal use. The Loge disk controller uses an indirection table (i.e., block-number map) to store the physical locations of logical blocks. In order to be able to recover this map, Loge includes the logical block number and a timestamp in the headers of the sectors being written.

Loge is a specific approach to improving performance that is more tightly bound than LD to specific hardware, as timely information is needed about the current position of the disk head. On the other hand, LD is a more general approach that is not bound to particular hardware, nor to a narrow class of software solutions. LD allows for substantially different implementations of its interface, which offers greater flexibility. In particular, the mechanism used to improve performance can be changed easily. For example, instead of using a log-structured approach, an LD implementation could use an update-in-place strategy or Loge's strategy. A log-structured LD could also use a heuristic similar to Loge's when choosing an empty segment to fill, thus integrating Loge's approach with that of LFS.

With respect to reorganization of disk layout LD and Loge exhibit another significant difference. Loge's write strategy makes it likely that logically related blocks get scattered over the disk. This behavior is somewhat similar to that of log-structured file systems. To prevent a loss of performance on sequential reads Loge, like log-structured file systems, needs to reorganize the data on disk. However, unlike log-structured file systems, Loge has to reorganize data based on the I/O stream only. This poses a serious problem, as in general it is not feasible to detect only from the block-level trace which blocks are related to each other. For example, if many different applications write concurrently to the same disk unit, the Loge disk controller cannot see which writes were issued by which application.

Another difference worth mentioning is that the recovery model of any *log-structured* LD implementation will be weaker than Loge's: Loge guarantees recovery up to the very last block successfully written, while an LLD guarantees recovery up to the last segment successfully written. This risk of losing more data is inherent to any approach that collects data for long contiguous writes to improve the effective use of the disk's bandwidth. There is an obvious trade-off: smaller segments diminish the potential loss, but also the potential performance gain. One could also take additional measures to alleviate this risk. For example, one could use NVRAM or ensure that on power failure each disk has enough power left to write the last segment.

In Loge the logical block numbers and timestamps of physical disk blocks are written into the sector headers; recovery therefore requires reading the whole disk. As a consequence, recovery in our LLD implementation is at least one order of magnitude faster than in Loge, since LLD only reads the segment summaries.

Mime [Chao et al. 1992] extends the work on Loge by offering a rich set of transaction-like capabilities. The most interesting aspect of Mime is its support of provisional writes by the use of so-called *visibility groups*. Visibility groups form a mechanism that can be used for controlling the mutual isolation of concurrent transactions. The LD interface has no direct counterpart for visibility groups. File systems using LD can implement isolation control by using atomic recovery units and a primitive that would swap the physical addresses of two logical blocks.

Mime recovers faster than Loge and at about the same speed as LLD, although recovery requires more than one revolution per cylinder for Mime, while one revolution per cylinder should suffice for LLD. Even though Mime writes a stream of individual blocks, just as Loge, it cannot offer the same recovery guarantees, as Mime has an operation log that is written to disk at every *sync*. Thus, Mime's recovery model is weaker than Loge's one, as is also the case for LLD.

5.3 Other related work

Akyürek and Salem describe an adaptive UNIX disk device driver [Akyürek and Salem 1993], which shares some ideas with work done by Vongsathorn and Carson [Vongsathorn and Carson 1990]. The driver periodically reorganizes the layout of blocks on the disk based on estimated reference frequencies, which are acquired by monitoring the stream of disk accesses. It copies frequently referenced blocks to reserved space near the center of the disk to reduce seek times. Measurements show that the adaptive driver reduces seek times by more than half and reduces response time significantly. As LD can rearrange blocks dynamically, the proposed scheme can be applied to LD too.

Baker et al. analyzed the impact of NVRAM on log-structured file systems [Baker et al. 1992]. They concluded that with 0.5 Mbyte of NVRAM the number of partially written segments can be reduced considerably; the number of disk accesses can be reduced by about 20% and on heavily used file systems it can even be reduced by about

90% [Baker et al. 1992]. We expect that similar results can be obtained for LLD.

Carson and Setia present an analytical derivation of the optimal segment size in log-structured file systems [Carson and Setia 1992]. They show that large segments are good for write performance, but can have an adverse effect on read performance. Based on an analytical model of a disk they conclude optimal segment sizes range between 52 Kbyte and 74 Kbyte, depending on the disk parameters. Our measurements for MINIX LLD show that 512-Kbyte segments are unnecessary large; using 128-Kbyte segments MINIX LLD achieves the same write performance. However, using 64-Kbyte segments we measured serious performance degradation for writes.

AIX Version 3 uses *database memory* to ensure that the AIX file system is always in a consistent state [Chang et al. 1990]. Database memory are files in virtual memory with the additional implicit properties of access serializability and atomic update. The database memory is implemented using a log to record the begin and the end of a transaction. All the file system metadata reside in database memory; therefore all the updates are treated as part of a transaction. Database memory is a mechanism that can be built on top of LD using LD's atomic recovery units.

The Echo [Hisgen et al. 1990] and Episode file systems [Chutani et al. 1992] are two recent file systems that use logging to increase reliability, to obtain good performance, and to restart quickly. Episode is a traditional UNIX file system that logs changes to file system metadata only. Echo logs all modifications: those to file system metadata as well as application data. Both file systems build on earlier file system that use logging, such as Cedar [Hagmann 1987] and Alpine [Brown 1985].

Seltzer et al. describe a log-structured implementation of 4.4BSD [Seltzer et al. 1993]. This particular implementation is more robust than Sprite LFS. The authors also report on a head-to-head comparison between FFS [McKusick et al. 1984], EFS [McVoy and Kleimann 1991], and BSD-LFS. The results show that EFS often provides comparable and sometimes superior performance to BSD-LFS due to BSD-LFS's cleaner competing for the disk arm. The authors, however, point out that BSD-LFS can be extended to incorporate additional functionality such as embedded transactions and versioning. Furthermore, BSD-LFS shows better performance in case of many small writes.

Van Renesse et al. describe a radically different organization for file systems, called the *Bullet* server, that achieves high throughput and low delay [Van Renesse et al. 1989]. Instead of storing files as sequences of disk blocks, each Bullet file is stored contiguously. Furthermore, Bullet files are immutable. This approach achieves high performance, but it has also has a number of disadvantages. As files are stored contiguously on disk and in main memory, it is impossible to cache files larger than the server's main memory or to cache multiple large files at the same time.

5.4 Future work

We have argued that LD provides a clean, flexible, and efficient interface for organizing file systems. MINIX LLD demonstrates that at least one file system can profit from LD and that LD can be implemented efficiently. We are in the process of incorporating LLD into the Mach 3.0 kernel and changing the UNIX server to use LLD. We are also designing another implementation of LD that stores data blocks at fixed disk locations and metadata in a log, similar to the related work we described in the previous section.

Using our Mach implementation we will be able to explore a number of issues in greater depth. First, we will be able to perform a head-to-head comparison with FFS, as the UNIX server is based on 4.3BSD. Second, we will be able to measure LLD's performance under different work loads. Third, we will able to determine how effective caching of the block-number map will be. Fourth, we will be able to experiment with different disk organizers and cleaners.

We are also considering a number of extensions to the LD interface. One extension would be to allow LD to export *variable-sized blocks*. This would allow file systems to write blocks of any size. However, this extension would be difficult to implement in a non-log-structured implementation.

Another extension we are considering is *offset addressing*, where lists could be indexed as arrays. For example, if we combine an implementation of the LD interface with an MS DOS file system [Tanenbaum 1987], we could eliminate the duplication of information in the File Allocation Table and LD's block-number map. A similar optimization could be applied to a UNIX file system: blocks of a file could be addressed by their offset from the first block in the list corresponding to the file, which eliminates the need for indirect blocks. Offset addressing also allows compact implementations of B-trees and their variants; it makes it possible to improve their branching factor considerably, as offset addressing allows all children of an index node to be addressed using only one address stored in the index node instead of one address per child.

The current interface does not support concurrent atomic recovery units, which is necessary for multithreaded file systems. We are considering various extensions to remedy this. For example, each operation could an atomic recovery unit identifier as an argument; *BeginARU* would generate these identifiers.

Finally, we are considering a primitive *SwapContents* that swaps the physical addresses of two logical blocks. Such a primitive would be useful for implementing transactions and multiversion data storage: new versions of blocks can be installed atomically without losing the old versions.

6. Conclusions

This paper presented the Logical Disk abstraction (LD), a log-structured implementation of LD, and new implementation techniques for log-structured file systems.

We have argued that disk and file management can and should be separated to make file systems simpler and more flexible. LD demonstrates that a clean, flexible, and efficient separation is possible. The LD interface provides a new abstract interface to the disk that is designed to support multiple file systems and to be implemented efficiently in multiple ways. It provides logical block numbers, block lists, atomic recovery units and multiple block sizes.

We have designed, built, and measured a log-structured implementation of the LD interface, called LLD. LLD is inspired by Sprite LFS [Rosenblum and Ousterhout 1992]. The key properties of LLD are: a clean separation of file and disk management, long contiguous writes, a simple and fast recovery scheme, and a new approach to writing partially full segments.

Not all of these properties are brand new, but the combination of these properties in LLD gives it a number of advantages over Sprite LFS and Loge. First, using LLD any file system can easily be turned into a log-structured file system, as demonstrated by MINIX LLD. Second, multiple different file systems can share the benefits of LLD. Third, cascading updates that can occur in Sprite LFS do not occur with LLD, which results in a performance and implementation benefit. The main disadvantage of LLD is that it requires a significant amount of main memory; LLD adds about 5% to 10% to the cost of a disk for the main memory it requires.

Unlike Loge, LLD can be used with existing disks. More importantly, LLD can do a better job of physically clustering blocks than Loge, as the file system can specify the logical relationship between blocks. Furthermore, Loge is not log-structured; LLD will show better performance when disk traffic is dominated by writes, which is likely when file systems have large main memories. A log-structured LLD could use Loge's approach of reducing disk seeks when choosing empty segments to fill, thus integrating Loge's approach with that of Sprite LFS. Finally, recovery with LLD is much faster than with Loge, as Loge reads the whole disk during recovery, whereas LLD only reads the segment summaries.

As more operating systems are starting to support multiple file and database systems, a modular and efficient approach to structuring file systems is vital. We are currently implementing LLD in the Mach 3.0 kernel; we hope to demonstrate that multiple user-level services, such as the UNIX server, can successfully benefit from LD, taking the trend of modularized operating systems one step further.

Acknowledgments

We thank Henri Bal, Kees Bot, Mike Burrows, Leendert van Doorn, Fred Douglis, Sanjay Ghemawat, Richard Golding, Bob Gruber, John Guttag, Philip Homburg, Kirk Johnson, Barbara Liskov, James O'Toole, John Ousterhout, Hans van Staveren, Andy Tanenbaum, Carl Waldspurger, John Wilkes (our "shepherd"), Quinton Zondervan, and the anonymous referees for their comments on drafts of this paper. We thank Eugene Belostotsky and Tom Pinckney for their work on the Mach implementation of LD.

References

[Akyürek and Salem 1993] Akyürek, S., and Salem, K., "Adaptive Block Rearrangement Under UNIX," *Proc. USENIX 1993 Summer Conference*, pp. 307-321, Cincinnati, OH, June 1993.

[Baker et al. 1992] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M., "Non-Volatile Memory for Fast, Reliable File Systems," *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM SIGPLAN Notices, Vol. 27, No. 9, pp. 10-22, Boston, MA, Oct. 1992.

[Brown et al. 1985] Brown, M.R., Kolling, K.N., and Taft, E.A., "The Alpine File System," *ACM Transactions on Computer Systems*, Vol. 3, No. 4, pp. 261-293, Nov. 1985.

[Burrows et al. 1992] Burrows, M., Jerian, C., Lampson, B., and Mann, T., "On-line Data Compression in a Log-structured File System," *Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM SIGPLAN Notices, Vol. 27, No. 9, pp. 2-9, Boston, MA, Oct. 1992.

[Carson and Setia 1992] Carson, S., and Setia, S., "Optimal Write Batch Size in Log-structured File Systems," *Proc. USENIX File System Workshop*, pp. 79-91, Ann Harbor, MI, May 1992.

[Chang et al. 1990] Chang, A., Mergen, F., Rader, R.K., Roberts, J.A., and Porter, S.L., "Evolution of Storage Facilities in AIX Version 3 for RISC System/6000 Processors," *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 105-110, Jan. 1990.

[Chao et al. 1992] Chao, C., English, R., Jacobson, D., Stepanov, A., and Wilkes, J., "Mime: a High Performance Parallel Storage Device with Strong Recovery Guarantees," HPL-CSP-92-9 rev 1, Hewlett-Packard Laboratories, CA, Nov. 1992.

[Chutani et al 1992] Chutani, S., Anderson, O.T., Kazar, M.L., Leverett, B.W., Mason, W.A., and Sidebotham, R.N., "The Episode File System," *Proc. USENIX Win-*

ter Conference 1992, pp. 43-60, San Francisco, CA, Jan. 1992.

[Custer 1993] Custer, H., “Inside Windows/NT,” Microsoft Press, Redmond, WA, 1993.

[English and Stepanov 1992] English, R.M., and Stepanov, A.A., “Loge: a Self-Organizing Disk Controller,” *Proc. USENIX 1992 Winter Conference*, pp. 237-251, San Francisco, CA, Jan. 1992.

[Golub et al. 1990] Golub, D., Dean, R., Forin, A., and Rashid, R., “UNIX as an application program,” *Proc. USENIX 1990 Summer Conference*, pp. 87-95, Anaheim, CA, June 1990.

[Hagmann 1987] Hagmann, R., “Reimplementing the Cedar File System Using Logging and Group Commit,” *Proc. of the 11th Symposium on Operating System Principles*, pp. 155-162, Austin, TX, Nov. 1987.

[Hisgen et al. 1990] Hisgen, A., Birrell, A.D., Jerian, C., Mann, T., Schroeder M., and Swart, C., “Granularity and Semantic Level of Replication in the Echo Distributed System,” *IEEE TOCS Newsletter*, Vol. 4, No. 3, pp. 30-32, 1990.

[IEEE 1990] IEEE, “POSIX - Part 1: System Application Program Interface (API) [C Language]”, IEEE Std 1003.1-1990, Jan. 1990.

[McKusick et al. 1984] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S., “A Fast File System for UNIX,” *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 181-197, Aug. 1984.

[McVoy and Kleimann 1991] McVoy, L., and Kleimann, S., “Extent-like Performance from a UNIX File System,” *Proc. USENIX 1991 Winter Conference*, pp. 33-44, Dallas, TX, Jan. 1991.

[Ousterhout 1990] Ousterhout, J., “Why aren’t Operating Systems Getting Faster as fast as Hardware?,” *Proc. USENIX 1990 Summer Conference*, pp. 247-256, Anaheim, CA, June 1990.

[Ousterhout and Douglis 1989] Ousterhout, J., and Douglis, F., “Beating the I/O Bottleneck: A Case for Log-structured File Systems,” *Operating Systems Review*, Vol. 23, No. 1, pp. 11-28, Jan. 1989.

[Rosenblum 1992] Rosenblum, M., “The Design and Implementation of a Log-structured File System,” Report No. UCB/CSD 92/26 (Ph.D. thesis), University of California, Berkeley, June 1992.

[Rosenblum and Ousterhout 1990] Rosenblum, M., and Ousterhout, J.K., “The LFS Storage Manager,” *Proc. USENIX 1991 Summer Conference*, pp. 215-324, Anaheim, CA, June 1990.

[Rosenblum and Ousterhout 1992] Rosenblum, M., and Ousterhout, J.K., “The Design and Implementation of a Log-structured File System,” *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 26-52, Feb. 1992.

[Ruemmler and Wilkes 1993] Ruemmler, C., and Wilkes, J., “UNIX Disk Access Patterns,” *Proc. USENIX 1993 Winter Conference*, pp. 405-420, San Diego, CA, Jan. 1993.

[Seltzer 1992] Seltzer, M., “File System Performance and Transaction Support,” Report No. UCB/CSD (Ph.D. thesis), University of California, Berkeley, Dec. 1992.

[Seltzer et al. 1993] Seltzer, M., Bostic, K., McKusick, M.K., and Staelin, C., “An Implementation of a Log-Structured File System for UNIX,” *Proc. USENIX 1993 Winter Conference*, pp. 201-220, San Diego, CA, Jan. 1993.

[Solworth and Orji 1991] Solworth, J.A., and Orji, C.U., “Distorted Mirrors,” *Proc. First International Conference on Parallel and Distributed Information Systems*, pp. 10-17, Miami Beach, FL, Dec. 1991.

[Tanenbaum 1987] Tanenbaum, A.S., “Operating Systems: Design and Implementation,” Prentice-Hall Inc., Englewood Cliffs, NJ, 1987.

[Van Renesse et al. 1989] Van Renesse, R., Tanenbaum, A.S., and Wilschut, A., “The Design of a High-Performance File Server,” *Proc. of the Ninth International Conference on Distributed Computing Systems*, pp. 22-27, Newport Beach, CA, June 1989.

[Vongsathorn and Carson 1990] Vongsathorn, P., and Carson, S.D., “A System for Adaptive Disk Rearrangement,” *Software—Practice & Experience*, Vol. 20, No. 3, pp. 225-242, Mar. 1990.