Logical Disk: A Simple New Approach to Improving File System Performance*

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

Making a file system efficient usually requires extensive modifications. For example, making a file system log-structured requires the introduction of new data structures that are tightly coupled with the general file system code. This paper describes a new organization for file systems, using a Logical Disk (LD); LD defines a simple new interface that separates file management and disk management. The interface simplifies the implementation of file systems and also improves performance. An implementation of a POSIX-compliant file system using LD confirms the benefits of this new organization. By trading main memory for a clean separation between file and disk management, LD allows file systems to achieve high performance.

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

1. INTRODUCTION

The performance of processors is increasing faster than that of disks [Ousterhout and Douglis 1989; Ousterhout 1990]. Therefore, more applications are expected to become I/O bound. Fortunately, the effect of the I/O bottleneck can be reduced by making more effective use of the full disk transfer rate. In most file systems the effective transfer rate is only a small percentage of the raw transfer rate, as most of the time spent in accessing disks is used to perform seeks.

One approach to solving the I/O bottleneck has been explored in the Sprite Log-structured File System (LFS), which increases the effective use of the disk bandwidth by writing dirty blocks to the disk in a single contiguous write [Ousterhout and Douglis 1989; Rosenblum and Ousterhout 1992]. This approach works because current file caches are large enough to satisfy most read requests; as a result, disk traffic is dominated by writes. Another approach to the I/O bottleneck has been explored by English and Stepanov; they have designed an intelligent disk controller (called Loge) that minimizes seek time and rotational delay for reading and writing individual blocks [English & Stepanov 1992]. When a block is written to disk, the Loge disk controller chooses an optimal disk location by selecting an unused physical block closest to the current position of the disk head.

In this paper we introduce a new approach to solving the I/O bottleneck that we call *Logical Disk (LD).* LD provides a simple new abstract interface to the disk that uses *logical block-numbers* and *lists of blocks*. The LD inter-

* This paper appears as technical report IR-325 at the Vrije Universiteit and as MIT/LCS/TR-566 at MIT.
† This work was partly 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.

face is powerful enough to simplify the efficient implementation of many file systems, yet abstract enough to allow for different implementations of the interface. LD defines an interface where the file system is responsible for file management and LD is responsible for disk management.

Separating these two concerns improves the structure and overall performance of a file system. To validate this we built a log-structured implementation of LD that is based on the ideas used in Sprite LFS. We combined this implementation of LD with a POSIX-compliant [IEEE 1990] file system, MINIX [Tanenbaum 1987]; with almost no modifications to MINIX we were able to make it log-structured. The resulting file system, MINIX LD, indeed exhibits the same performance characteristics as Sprite LFS. In addition, we added compression with no modifications to MINIX and very few modifications to LD. If we removed disk management from the MINIX code, MINIX LD would outperform Sprite LFS, as it would write fewer blocks. MINIX LD requires more memory than Sprite LFS, but it achieves a clean separation between file and disk management as a result.

The outline of the rest of the paper is as follows. Section 2 describes the LD interface and an LD implementation. Section 3 provides a performance evaluation of MINIX LD. In Section 4, we compare MINIX LD with Sprite LFS, compare LD with Loge, and discuss the LD interface. In Section 5 we summarize our results and conclusions.

2. LOGICAL DISK

In this section we briefly describe the two primary abstractions supported by the LD interface (in Section 4 we discuss other abstractions that we are considering). We then describe a log-structured implementation of the LD interface with which we conducted our experimental evaluation.

2.1 The LD Interface

The two primary abstractions in the LD interface are *logical block-numbers* and *block lists*. 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 always address blocks by their logical addresses, which do not change, even if LD changes their physical locations.

Using logical block-numbers has three advantages. First, the file system is simple, as LD can perform disk management transparently. For example, LD can reorder blocks by simply updating the block-number map. Second, it is also easy to modify disk management policies: LD can use either a log-structured approach or a traditional update-in-place approach without the file system's knowledge. Third, it has performance benefits, as cascaded updates to file system data structures can be avoided. For example, Sprite LFS stores physical disk addresses in indirect blocks, double-indirect blocks, and i-nodes. If a data block is updated or moved in Sprite LFS, its physical address changes; therefore, 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 cascaded updates do not happen.

In order to effectively separate disk and file management, LD needs to know the relationship between blocks. To allow file systems to express these relationships, LD supports the simple but powerful abstraction of ordered *lists* of logical blocks. LD can physically cluster the blocks according to the order in which they appear in such a list. For example, when a file system puts the blocks of a file (including i-node and indirect blocks) on a list, LD tries to place those blocks near each other physically; as a result, unnecessary disk seeks are 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; it could correspond to some other order that the file system knows will be the order in which they will be read. With lists the use of cylinder groups [McKusick et al. 1984] is unnecessary. In addition, lists can be used to perform read-ahead.

2.2 An LD Implementation

Different implementations of the LD interface are possible; we describe one particular implementation based on the ideas used in Sprite LFS. This implementation, which we will also refer to as LD, is based on the assumption that reads are absorbed by the file system cache and therefore disk traffic is dominated by writes. LD, like LFS, writes dirty file system blocks in a single write. It divides the disk into large, fixed-size *segments* and writes a segment sequentially from beginning to end. The end of each segment contains a *segment summary* that records information for each block in the segment. A *segment cleaner* periodically goes over the disk and cleans segments using the information stored in the segment summaries, so that empty segments become available. To determine which segments to clean, LD maintains a *segment usage table* that records the number of live bytes in each segment. In addition to cleaning segments, the segment cleaner also reorders live data blocks on disk according to list information. Table 1 summarizes the data structures used by LD.

Data Structure	Location	Description
Block-number map	Main memory	The map stores for each logical block its physical address, its successor in its list, its length, and whether it is compressed.
List table	Main memory	The table stores the numbers of the first and last logical block of each list.
Segment	Disk	A segment is a fixed-size structure on disk 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 it is written in a single operation when the segment is flushed.
Segment summary	Disk	A summary stores the size of the summary and the number of logical blocks in the segment. For each physical block in the segment, the summary stores its logical block-number, its time stamp, its length, and whether it is compressed. In addition, it stores link tuples and the number of link tuples.
Segment usage table	Main memory	The table stores the number of live bytes in each segment.

Table 1. Data structures maintained by a log-structured implementation of LD that supports compression.

When the file system allocates a block for a file, LD updates the list information in the block-number map and the list table, and for each successor field updated it appends a link tuple <time stamp, list identifier or block number, block number of successor> to the in-core segment summary.

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

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

To demonstrate the flexibility of LD, we have implemented compression, which also has been done for Sprite LFS [Burrows et al. 1992]. Although any compression algorithm will work with our approach, we have chosen to use an algorithm due to Wheeler for its simplicity and performance; the algorithm is described in [Burrows et al. 1992]. Using LD, a file system can transparently use compression to make more effective use of disk space. LD compresses user data and file system data structures (e.g., i-nodes), but not its own data structures.

To support compression LD internally uses variable-sized blocks. LD compresses blocks into the in-core segment 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, LD checks in the block-number map whether the block is compressed. If so, LD uncompresses it before returning it to the file system. LD currently compresses all blocks, but it may be a better strategy to only compress cold files during cleaning; under such a strategy writes and reads of recently written data can be performed at the maximum disk bandwidth instead of at the bandwidth of the compression algorithm.

Storing the block-number map requires a substantial amount of main memory: without compression, each logical block uses three bytes for its physical block address and three bytes for its successor. With a 4-Gbyte disk and 4-Kbyte blocks, the block-number map requires 6 Mbyte of memory. To support compression at most two bytes are needed to store the length and an additional byte is needed for the physical address; in this case the block-number map requires 9 Mbyte of main memory. The marginal cost, however, of 9 Mbyte of main memory is low compared to the total cost of a complete file server. Storing the list table takes eight bytes per file, and storing the segment usage table takes three bytes per segment.

Without compression, each physical block requires seven bytes in the segment summary, three for its logical number and four for its time stamp. In addition, link tuples are stored in the segment summary, which is done with 12 bytes per tuple (but it can be done with 10 bytes). 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 LD, 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.

Rosenblum and Ousterhout describe a number of policies for selecting and cleaning segments using the segment usage table, and all of these can be used for LD as well [Rosenblum and Ousterhout 1992]. Using the lists, LD can physically cluster related blocks. Currently, LD uses a very simple clustering strategy: when it copies blocks from a segment that is being cleaned, it reorders the blocks so that related blocks are near each other physically. LD also removes old link tuples from the segment summaries during cleaning.

To reconstruct the block-number map and the segment usage table after a system failure, LD reads all of the segment summaries in a single sweep over the disk. For each disk block registered in a segment summary, LD uses its time stamp to determine whether it is the most recent version of a logical block. This information is used to reconstruct the segment usage table and the physical address part of the block-number map. Similarly, LD uses the link tuples in the segment summaries to restore the most recent links in the list table and the block-number map.

3. EXPERIMENTAL EVALUATION

We have implemented LD as a user-level process and linked it through a message passing interface to the MINIX file system. LD writes and reads blocks from a disk partition using UNIX system calls. Although the file system and LD run within the same process, they communicate only through messages; in principle, both the file system and LD could run in separate processes.

In order to let MINIX work efficiently with LD, we made four changes* to it. First, MINIX notifies LD when it allocates a new block for a file; it tells LD to add the block to the list corresponding to the file. Second, when MINIX frees a block it notifies LD that the block is free. Although this change is not strictly necessary, it allows LD to do a better job of segment cleaning; without it, LD can only discover that a logical block is dead when the file system reallocates it. Third, upon a *sync* MINIX tells LD 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.

To measure MINIX LD's performance we ran the same microbenchmarks as Rosenblum and Ousterhout [Rosenblum and Ousterhout 1990]. 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. To evaluate the performance of MINIX LD 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 LD ran as user-level processes that used the raw disk interface provided by SunOS. For the experiments, MINIX LD used a 0.5-Mbyte segment 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, the file cache was flushed after each phase in the experiments. Both MINIX and MINIX LD 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 2346 Kbyte/s on this configuration. No cleaning occurred during the experiments.

Table 2 shows the results of running the benchmark for small file I/O in files per second. The numbers for MINIX and MINIX LD include neither the overhead of the application itself, nor that of the pipes between the application and the file system. File creation of files is faster in MINIX LD than in MINIX because MINIX LD collects many changes in a single write. File reading in MINIX LD has the same speed as in MINIX because reads are per-

* In addition, we fixed three performance bugs in MINIX itself after we started taking measurements.

formed sequentially in both file systems. File deletion in MINIX and MINIX LD have similar performance, since deletions take the same amount of work in both systems.

With one exception, the file operations in MINIX and MINIX LD for 10-Kbyte files are more efficient than those for 1-Kbyte files due to the nature of the benchmark and an artifact of the MINIX implementation. MINIX uses a linear search algorithm to search directories; thus, the work per file is linear in the number of files created in an experiment. The exception is for file creation in MINIX; this is explained by the large overhead of writing sequential blocks, as discussed below.

The numbers in the table for SunOS are worse for creation and deletion than for the two MINIX file systems, since SunOS performs the operations synchronously. SunOS reads are better because SunOS has more disk bandwidth available: MINIX and MINIX LD depend on SunOS to access the disk.

File System	10,000 1-Kbyte file (file/s)			1,000 10-Kbyte file (file/s)		
	Create	Read	Delete	Create	Read	Delete
MINIX LD	64	76	138	119	97	610
MINIX	38	74	144	25	100	758
SunOS	13	140	25	16	108	24

Table 2. Performance results for creating, reading, and deleting 10,000 1-Kbyte and 1,000 10-Kbyte files in one directory. Both MINIX file systems ran as user-level processes that used a raw SunOS disk partition. Read performance is less significant, since the assumption in log-structured file systems is that reads are mostly absorbed by the file system cache. The numbers are in files per second (higher numbers are better).

Table 3 shows the results for large file I/O in Kbytes per second. Like Sprite LFS, MINIX LD shows excellent performance on all writes, as all file writes are turned into sequential disk writes; MINIX LD uses 79% of the available bandwidth. MINIX, on the other hand, uses only 13% of the available bandwidth. MINIX's throughput is so low because the overhead of the disk controller and SunOS is high enough that 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 324 Kbyte per second); the same phenomenon was detected in the original Sprite file system [Rosenblum 1992]. If we changed MINIX's block allocation strategy, MINIX should achieve similar performance on sequential writes as MINIX LD.

File System	Write sequential	Read Sequential	Write Random	Read Random	Read Sequential
MINIX LD	1861	1310	1951	443	426
MINIX	316	1454	324	222	1460
SunOS	2938	3804	640	442	3869

Table 3. Performance results for writing and reading a 80-Mbyte file (in 8-Kbyte chunks) for MINIX LD and MINIX; for SunOS a 300-Mbyte file was used to reduce the effects of its larger file cache. Both MINIX file systems ran as user-level processes that used a raw SunOS disk partition. Read performance is less significant, since the assumption in log-structured file systems is that reads are mostly absorbed by the file system cache. The numbers are in Kbyte/s (higher numbers are better).

MINIX achieves higher throughput than MINIX LD on sequential reads because it uses prefetching, which we disabled for MINIX LD. The performance of random reads for MINIX LD 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 LD, 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 (with one exception) than the MINIX file systems: it has more disk bandwidth available, since MINIX depends on SunOS to access a raw disk partition. The exception is random reads; SunOS performs about the same as MINIX LD. This anomaly is most likely due to unsuccessful read-ahead. For random writes the performance of SunOS is worse than MINIX LD, because MINIX LD turns random writes into sequential writes.

In addition to running the microbenchmarks, we measured the time for MINIX LD to recover. 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 LD that does not support lists. Different runs of the benchmark have shown that indeed there is no overhead during reading or writing. There is only overhead during block allocation and deallocation; during the create and delete phases the overhead for maintaining lists was approximately 15%.

We also measured the throughput of MINIX LD with compression on */vmunix;* the write throughput was 1611 Kbytes per second, and the read throughput was 749 Kbytes per second. The write throughput is within 15% 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 the reading and decompressing of a disk block; however, given our assumption that reads will be absorbed by the file system cache, the lower read throughput is not significant. Therefore, compressing every block is likely to be a good choice in systems with sufficiently large file caches; in systems with small file caches, it would be better to only compress cold files during cleaning.

The measurements show that MINIX LD indeed makes more effective use of the disk bandwidth than MINIX. Like Sprite LFS, MINIX LD is CPU-bound and not disk-bound, so its performance will scale with processor speed. If we ran the unmodified MINIX file system on a faster processor, its performance would not improve considerably, as most of the time the processor would be idle waiting for the disk. 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.

4. COMPARISON AND DISCUSSION

In the previous section we discussed the measured performance of MINIX LD; we did not do a head-to-head performance comparison of MINIX LD and Sprite LFS. This section begins with a discussion of Sprite LFS and MINIX LD, and argues that MINIX LD would perform better than Sprite LFS if disk management were removed from MINIX. We then compare LD with Loge; we conclude with a discussion of other abstractions for the LD interface. The work done by Douglis, Rosenblum, and Ousterhout has generated a number of other results that also apply to a log-structured implementation of LD [Baker et al. 1992; Carson and Setia 1992]; we will not discuss them.

4.1 Comparison with Sprite LFS

Table 4 compares Sprite LFS and MINIX LD on a number of issues, the most important of which we will 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 when it flushes a segment). The i-nodes must be written out because the disk addresses and the modifications 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 small and is denoted by ε; the actual cost of file creation is therefore $3 + 2\varepsilon$ blocks.

For the same operation MINIX LD writes the updated directory block, two dirty i-nodes, one block in the block bitmap, and one in the i-node bitmap. The bitmaps are used by MINIX to manage free disk space; since LD manages the disk, these data structures are unnecessary. Thus, if we made some additional small changes to MINIX the total would be three (the data block and the two i-nodes) instead of five blocks. MINIX LD avoids cascaded updates, but still writes the i-nodes in order to make the modification time recoverable, as required by the POSIX standard. As we discuss in Section 4.3, we could reduce the cost of writing an i-node to ε; then the total would be $1 + 2\varepsilon$ blocks.

Issue	Comparison
Reading a block	Performance is equal for Sprite LFS and MINIX LD.
Creating or deleting a file	Sprite LFS writes $3 + 2\varepsilon$ blocks; MINIX LD writes 5 ($1 + 2\varepsilon$) blocks.
Overwriting a block	Sprite LFS writes $2 + \varepsilon$, $3 + \varepsilon$ or $4 + \varepsilon$ blocks; MINIX LD always writes 2 ($1 + \varepsilon$) blocks.
Appending a block	Sprite LFS writes $2 + \varepsilon$, $3 + \varepsilon$ or $4 + \varepsilon$ blocks; MINIX LD usually writes 3 ($1 + \varepsilon$) or 4 ($2 + \varepsilon$) blocks. MINIX LD only writes 5 ($3 + \varepsilon$) 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 LD requires only minor modifications.
Memory requirements	MINIX LD uses an additional 4 to 6 Mbyte main memory for its data structures.
Cleaning and clustering	MINIX LD requires less segment cleaning than Sprite LFS, since it fills segments with more user data. Reorganization is also easier with MINIX LD.
Recovery	Sprite LFS may recover somewhat faster, but requires checkpoints during normal operation.

Table 4. Comparison between Sprite LFS and MINIX LD. The cost in Sprite LFS for creating and overwriting will be lower when a segment is shared by many files; the cost for overwriting will also be lower when many data blocks belonging to the same file are written at once. The costs for MINIX LD for creating and overwriting can be lowered at the cost of additional changes to MINIX, as indicated by the values in parentheses.

The overhead of cascaded 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 LD there are no cascaded updates, and none of these blocks need to be rewritten.

A more precise comparison would also take into account the fact that a segment can be shared by many files, and that many data blocks belonging to the same file can be written at once. For example, when Sprite LFS applies single-block updates to many different files, writing i-node map blocks will cost somewhat less than one block per update, since some i-node map updates will occur in the same i-node map block. As another 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 i-node map block, 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 LD for this last example is at most a few percent.

A disadvantage of MINIX LD is that it uses 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, a simple implementation of LD not supporting compression uses 6 Mbyte of main memory to store the block-number map (see Section 2). Assuming that each user file is on average 23.5 Kbytes, and that each i-node map entry is 12 bytes, the i-node map for a 4-Gbyte disk with 4-Kbyte blocks requires 2 Mbyte of main memory [Rosenblum 1992]. Thus, compared to Sprite LFS without compression, a simple implementation of MINIX LD needs between 4 and 6 Mbyte of additional main memory. This memory leads to a cleaner structure in the file system as well as higher performance. Furthermore, it was easy to add compression to MINIX LD and at a price of only 3 Mbyte extra.

Segment cleaning and physical clustering are more efficient in MINIX LD than in Sprite LFS. During cleaning and reorganization live data blocks will be moved, and their physical addresses will change. Unlike in MINIX LD, in Sprite LFS the related i-nodes, indirect blocks, and double-indirect blocks always have to be updated and moved as well. The same advantage in another disguise is that MINIX LD has to perform segment cleaning less frequently, as it fills each segment with more user data than Sprite LFS does.

Sprite LFS periodically stores the location 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 LD does not use checkpoint regions; therefore, recovery may take more time, since MINIX LD reads all of the segment summaries to reconstruct its block map. Assuming a 4-Gbyte disk and 4-Kbyte blocks, Sprite LFS reads up to about 500 blocks (2 Mbyte) for the i-node map and then reads all segments written since the last checkpoint, while MINIX LD reads in one sweep about 8,000 segment summaries; as shown by the measurements in the previous section, MINIX LD's recovery strategy is still quite fast.

4.2 Comparison with Loge

Loge [English and Stepanov 1992] improves the I/O performance of disks by having the disk controller minimize the time required for the reading and writing of a stream of individual blocks. When a block is written to disk, the Loge disk controller chooses an optimal disk location; i.e., an unused physical block closest to the current position of the disk head. The Loge disk controller uses an indirection table (i.e., block-number map) to store the physical locations of blocks. In order to be able to recover this map, Loge includes the logical block-number and a time stamp in the headers of the sectors being written.

Unlike Loge, LD is a software solution, which allows for different implementations of the interface. The particular mechanism used to improve performance can be changed easily. For example, instead of using a log-structured approach, LD could be based on an update-in-place strategy or a strategy similar to Loge's. 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.

A significant difference between LD and Loge is that Loge organizes data based only on the I/O stream. Since the LD interface provides a mechanism to convey information about relationships between blocks, LD can do better physical clustering than Loge. 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, and in general it is not feasible to detect only from the trace which blocks are related to each other.

In Loge the block-numbers and time-stamps related to physical disk blocks are written into the sector headers and recovery therefore requires reading the whole disk. As a consequence, recovery in LD is between one and two orders of magnitude faster than in Loge, since LD only reads the segment summaries.

4.3 More about the LD Interface

We are considering a number of extensions to the LD interface to increase the functionality and the performance of LD. One is to have LD export variable-sized blocks to file systems. This allows efficient storage of i-nodes and directories. For example, if MINIX used 32-byte blocks to store i-nodes, file creation in MINIX LD would require $1 + 2\varepsilon$ blocks to be written. For MINIX or any other current UNIX file system, however, taking advantage of variable-sized blocks would require substantial modifications to the general file system code. It would require only minor modifications to LD itself, since LD already uses variable-sized blocks internally.

Another useful extension is to allow file systems to write multiple blocks indivisibly. By using this extension a file system could treat the creation of a file and the update of its directory as a single operation. This would eliminate the need for consistency checks when running *fsck*. (Sprite LFS already eliminates the use of *fsck* by keeping log entries for directory modifications [Rosenblum and Ousterhout 1992].) In addition to making consistent file creation simpler, this extension allows file systems to export indivisible operations to user applications. Again, this extension would require minor modifications to our current implementation of LD.

Finally, we are considering *offset addressing*, where lists could be indexed as arrays. For example, if we combined an implementation of the LD interface with the MS DOS file system (FAT) [Schulman et al. 1990], we could eliminate redundancy in information in the FAT and LD. Similar optimizations could be applied to a UNIX file system; for example, we could eliminate double-indirect blocks. Offset addressing can also be used to improve the branching factor of B-trees and its variants.

5. CONCLUSION

This paper has presented a simple new approach to structuring file systems, using LD, that improves file system performance. The LD interface provides a new abstract interface to the disk that provides logical block-numbers and block lists. Although more research is needed to define a generic interface for a secondary memory server that supports different file systems (including databases) and different implementations, we believe that logical block-numbers, block lists, variable-sized blocks, indivisible writes, and offset addressing are key abstractions in such a generic interface.

In addition to introducing a new disk abstraction, we have designed, built, and measured a log-structured implementation of the LD interface. This log-structured LD incorporates ideas from Sprite LFS [Rosenblum and Ousterhout 1992] and Loge [English and Stepanov 1992]. Five key properties of this LD implementation are: large sequential writes, logical block-numbers, lists of blocks, simple and fast recovery, and a clean separation of file and disk management. The first two properties are not new, but the way LD puts these properties together with the new properties gives LD a number of advantages over Sprite LFS and Loge.

Compared to Sprite LFS, LD has two main advantages. First, LD makes it easy to make a file system log-structured, while the conversion to a log-structured file system usually requires extensive modifications to general file system code (see, for example, [Seltzer et al. 1993]); LD can also be used to turn non-UNIX file systems into log-structured ones. The mapping from logical numbers to physical addresses can be done transparently in a log-structured LD: the file system is unaware that LD divides the disk into segments, writes blocks collectively in a single disk operation, cleans segments, and performs physical clustering. To implement this functionality, LD requires a substantial amount of main memory.

Second, a UNIX file system using LD delivers better performance than Sprite LFS. The use of logical block-numbers prevents updates on data blocks from cascading to indirect, double-indirect, i-node, and i-node map blocks. LD also employs a new recovery strategy that prevents changes to the block-number map from resulting in a substantial number of additional writes. As a result, the number of disk writes resulting from a certain set of updates will be lower; thus, the net data transfer rate will be higher than in Sprite LFS.

Unlike Loge, LD can be used with existing disks. More importantly, LD does a better job of physically clustering blocks than Loge, as the file system can specify the relationship between blocks. Furthermore, Loge is not log-structured; log-structured LD therefore shows better performance when disk traffic is dominated by writes, which is the case when file systems have large main memories. A log-structured LD could use Loge's approach of minimizing disk seeks when choosing empty segments to fill, thus integrating Loge's approach with that of Sprite LFS. Finally, recovery with LD is between one and two orders of magnitude faster than with Loge, as Loge reads the whole disk during recovery, whereas LD only reads the segment summaries.

To demonstrate the validity of our approach, we have combined the MINIX file system with LD to turn it into a log-structured file system, MINIX LD; this required only a few small changes to MINIX. Measurements of MINIX, MINIX LD, and SunOS show that MINIX LD indeed has the performance characteristics of a log-structured file system.

Acknowledgments

We wish to thank Henri Bal, Kees Bot, Leendert van Doorn, Fred Douglis, Sanjay Ghemawat, Richard Golding, Bob Gruber, Philip Homburg, Kirk Johnson, Barbara Liskov, Hans van Staveren, Andy Tanenbaum, Carl Waldspurger, and Quinton Zondervan for their careful reading of drafts of this paper and their suggestions.

References

Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M., "Non-Volatile Memory for Fast, Reliable File Systems," *Proc. Fifth Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 10-22, Boston, MA, Oct. 1992.

Burrows, M., Jerian, C., Lampson, B., and Mann, T., “On-line Data Compression in a Log-structured File System,” *Proc. Fifth Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-9, Boston, MA, Oct. 1992.

Carson, S., and Setia, S., “Optimal Write Batch Size in Log-structured File Systems”, *Proc. File System Workshop* 1992, pp. 79-91.

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.

IEEE, “POSIX - Part 1: System Application Program Interface (API) [C Language]”, IEEE Std 1003.1-1990, January 1992.

McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S., “A Fast File System for UNIX,” *ACM Trans. Comp. Syst.*, Vol. 2, No. 3, pp. 181-197, Aug. 1984.

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

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, M., and Ousterhout, J.K., “The Design and Implementation of a Log-structured File System,” *ACM Trans. Comp. Syst.*, Vol. 10, No. 1, pp. 26-52, Feb. 1992.

Schulman, A., Michels, R.J., Kyle, J., Paterson, T., Maxey, D., and Brown, R., “Undocumented DOS,” Addison-Wesley, Reading, MA, 1990.

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.

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