Beating the I/O Bottleneck:
A Case for Log-Structured File Systems

John Ousterhout
Fred Douglis

Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, CA 94720

Abstract

CPU speeds are improving at a dramatic rate, while disk speeds are not. This technology shift suggests that many engineering and office applications may become so I/O-limited that they cannot benefit from further CPU improvements. This paper discusses several techniques for improving I/O performance, including caches, battery-backed-up caches, and cache logging. We then examine in particular detail an approach called *log-structured file systems*, where the file system's only representation on disk is in the form of an append-only log. Log-structured file systems potentially provide order-of-magnitude improvements in write performance. When log-structured file systems are combined with arrays of small disks (which provide high bandwidth) and large main-memory file caches (which satisfy most read accesses), we believe it will be possible to achieve 1000-fold improvements in I/O performance over today's systems.

The work described here was supported in part by the National Science Foundation under Presidential Young Investigator Award No. ECS-8351961 and Grant No. MIP-8715235.

1. Introduction

If a computer program or system consists of several components, and if some of the components are made much faster while leaving the other components unchanged, then the unimproved components are likely to become a performance bottleneck. This notion was first put forth by Gene Amdahl in the late 1960's [1], and has since come to be known as ''Amdahl's Law.''

Recent technology trends suggest that disk input and output may become such a bottleneck in the near future. CPU speeds are increasing dramatically along with memory sizes and speeds, but the speeds of disk drives are barely improving at all. Without new file system techniques, it seems likely that the performance of computers in the 1990's will be limited by the disks they use for file storage. For example, suppose that an application program today spends 10% of its time waiting for disk I/O. If CPU speed improves by a factor of 10 without any disk improvements, then the application will only run about 5 times faster and will spend about 50% of its time waiting for I/O. If CPU speed improves by another factor of 10, still without disk improvements, then the application will only run about twice again as fast, and will spend about 90% of its time waiting for I/O. The hundred-fold overall improvement in CPU speed will only result in a ten-fold improvement in application speed.

Fortunately, there are two technology trends that offer hope for beating the I/O bottleneck: large memories and disk arrays. First, as memories get larger and larger, it becomes possible to cache more and more file information in main memory and thereby eliminate many disk accesses. Second, disks are becoming much cheaper and physically smaller. This makes it practical to build disk systems containing tens or hundreds of drives (''disk arrays''). Disk arrays don't improve the performance of a single small access, but they offer much greater overall bandwidth and the possibility of many concurrent accesses.

This paper is a discussion of the technology trends related to I/O, the problems they may pose for future engineering and office applications, and a set of possible solutions. Our overall goal is to find ways of capitalizing on fast-moving technologies (CPU speed, memory size, disk array size) to compensate for slow-moving technologies (disk seek times) so that future systems will not be I/O-limited.

Section 2 considers the technology trends in more detail, with special emphasis on disk arrays. Section 3 uses information about file access patterns to compute a worst-case scenario for I/O requirements of the future. Section 4 discusses file caching; this is a well-known technique that is already implemented in several systems, but Section 4 argues that write traffic limits the performance improvements of caching alone. Sections 5 and 6 discuss two ways to improve file caches by reducing write traffic: battery backup and cache logging.

In Section 7 we make our most radical and detailed proposal, which is to restructure the file system so that its only representation of information on disk is in the form of a circular append-only log. Such a *log-structured file system* eliminates almost all seeks during writes, and thus may improve writing performance by an order of magnitude or more. The logging approach also makes it easier to utilize the additional bandwidth provided by disk arrays. When combined with a large main-memory file cache, the cache will satisfy almost all of the read requests while the log-structured disks will handle writes efficiently. Overall, we think that this approach can result in ten times better utilization of disk bandwidth than today's file systems; if used with disk arrays containing hundreds of disks, three orders of magnitude total improvement in file system performance may be possible. Log-structured file systems also have other advantages including faster crash recovery and automatic clustering of files written at the same time.

Section 8 describes other work with similarities to log-structured file systems, and Section 9 concludes the paper.

The success of the new techniques described in this paper may not be determined for some time. We have not implemented a log-structured file system (yet), and the hardware for which it is most suitable will not exist for several years. We cannot even say for sure that I/O will prove to be the bottleneck we suggest. Nonetheless, we hope that, by exposing the issues early, we can encourage researchers to experiment with these and other approaches to high-performance I/O. With luck, an acceptable solution will have been demonstrated before an I/O bottleneck ever manifests itself.

2. Technology Trends

When some technologies advance more rapidly than others, changes occur in the basic tradeoffs of system design, offering interesting problems and alternatives to system builders. We call such a change a *technology shift*. Engineering and office computing environments have witnessed at least two major technology shifts in recent years. The first shift occurred between 1975 and 1985, when memory sizes increased enormously while CPU and disk speeds hardly changed. This opened up new memory-intensive application areas such as bit-map graphics and laser printing, and made it possible to improve I/O performance by caching files in main memory. The second technology shift is underway now. Between 1985 and 1995, typical CPU speeds in engineering and office environments are likely to increase by factors of 100 to 1000. Memory sizes will also increase substantially, although probably by a factor of less than 100. In contrast, disk speeds (particularly seek times) seem unlikely to improve by more than a factor of 2 or 3 in the same time period. This shift will open up new CPU-intensive applications, but suggests that new I/O techniques will be needed to keep I/O from being a performance bottleneck.

Although disk drives have not improved much in performance, they have improved radically in cost and storage density. The trend in recent years is towards disk drives that are physically smaller and much cheaper, yet still have performance and capacity comparable to the bulkier drives they replace. We are now seeing the advent of disk arrays, where large high-performance reliable disk systems are created by combining many small inexpensive disks [12]. By 1995, we predict that disk arrays with hundreds or even thousands of disks will be standard products.

Disk arrays offer at least two potential performance advantages. First, a disk array permits much greater overall bandwidth than a single disk, if the array is organized to transfer to/from all its disks in parallel. Second, a disk array permits many more independent accesses per second, if there is enough work to keep many of the disks busy simultaneously. Unfortunately, disk arrays do not improve the basic disk latency for a given access; if an application makes sequential accesses to small unrelated pieces of data then it will not run any faster with a disk array than with a single disk.

Thus, the key question that this paper addresses is: how can we take advantage of the technologies that are improving most rapidly (CPU speed, memory size, disk array size), and use them to compensate for technologies that are not improving as rapidly (seek times)?

3. The File Access Problem

In this section we attempt to predict how today's file systems will behave if the technology predictions of Section 2 are accurate and if no changes are made in file system organization. Section 3.1 discusses file access patterns, and Section 3.2 estimates the I/O demands that might be placed on a computing system of the mid-1990's.

3.1. File Access Patterns

In order to understand the impact of technology changes on file system performance, one must consider how file systems are used by application programs. We think that file access patterns can be separated into three general classes: scientific computing, transaction processing, and engineering/office applications. Of these classes, the first two appear well-suited to disk arrays while the third does not.

The first class of file usage, which we call ''scientific processing'', is characterized by programs that read and write very large files sequentially. Many scientific applications and some engineering and database applications fall into this category. If sequential blocks of large files are spread evenly across the disks of an array (''striped''), then all the disks can be used simultaneously to read or write the files [7]. If the files are large enough, then the cost of seeks will be small compared to the cost of reading the data, and the file system's performance will scale with the size of the array. Thus if the CPU power and number of disks in an array scale at about the same rate (which appears likely for the next several years), then scientific applications should be able to scale smoothly in performance even though the disk latency does not improve.

We use the term ''transaction processing'' to refer to applications like airline reservations, automatic tellers, and point-of-sale terminals. These systems typically handle large numbers of concurrent requests, each of which makes a small number of disk accesses (for example, to reserve a seat or make a deposit). Transaction processing systems are usually measured by their throughput in transactions per second, rather than by the service time for individual transactions. Different requests usually involve different disk data, so it is possible to keep many disks busy simultaneously. This means that transaction processing systems will also scale smoothly as long as the number of disks in an array increases as fast as CPU power: the speed of an individual transaction will still be limited by latency, but the system's overall throughput will increase.

The third class of file usage, ''engineering/office applications,'' consists of programs that access large numbers of small files, such as the source files for a program or the font libraries for a laser printer. Most programs in UNIX environments fall into this class. These applications are similar to transaction processing systems in that they make small requests. However, they are different from transaction processing in that there isn't much concurrency: each application program makes a sequence of requests, and each CPU typically runs only one or a few application programs at once. Furthermore, the goal of increasing the CPU power in an engineering/office environment is to make individual programs run faster, not to increase the number of concurrent programs. As a result, these applications do not appear to benefit from disk arrays: files are short enough that striping doesn't help (most of the access cost is in the latency), and there isn't enough concurrency to occupy the disks of a large array.

For the rest of this paper we will focus on engineering and office applications. This is partly because it is less obvious how to take advantage of disk array technology for these applications than for the others, and partly because we have more experience with them than with the other applications. However, some of the results also apply to scientific applications.

3.2. How Many Disks per User?

Suppose that I/O rates generated by application programs scale evenly with CPU power. Several recent studies, plus our own measurements, suggest that the average I/O traffic generated by a single user on a 1-MIP engineering workstation is about 2.5-10 kbytes/sec. [8,11]. In the environment of the mid 1990's with 100-MIP workstations, each user might then generate 250-1000 kbytes/sec. of I/O traffic.

The average file size in engineering/office environments is only 3-4 kbytes [11,13]. With today's file organizations (e.g. 4.3 BSD UNIX [9]), two disk transfers are required for each file access: one to read or write file header information, and another to read or write the file's data. Typical disks today can only make about 30-40 transfers per second. Thus the effective disk bandwidth for a single disk is about

$$\text{((30-40 transfers/sec.) / (2 transfers/file)) * (3-4 kbytes/file)}$$
$$\approx \text{60 kbytes/sec.}$$

If file system organization and average file sizes don't change over the next five years, then the above calculations suggest that future systems will need 4-16 disks per user to keep up with I/O demands. This has two unpleasant implications. First, it suggests that disks would represent by far the greatest part of the cost of a computer system. Second, it requires the operating system to organize the file system so that it can keep 4-16 disks busy for each user: if a user's data ends up all on a single disk, then the file system will not be able to keep up with the user's I/O demands.

Of course, it's possible that I/O access patterns will change as CPU power increases. For example, the extra CPU power of future machines might be used primarily for managing the user interface (graphics, speech recognition, etc.) and thereby require very little additional I/O. Or, average file sizes might increase and thereby allow more efficient utilization of disk bandwidth; this could happen if, for example, future applications make heavy use of large images. To the extent that these changes occur, I/O demands will be less than we estimated above. On the other hand, CPU power might increase to more than 100 MIPs by the mid-1990s; if it increases to 500 MIPs, then as many as 20-80 disks per user might be required! Overall, we think it is plausible that disk requirements will be severe enough to present difficult problems both in terms of cost and in terms of keeping them all busy.

4. Solution #1: File Caching

File caching has been implemented in several systems to improve file I/O performance [6,9,10,14]. The idea in file caching is to retain recently-accessed disk blocks in main memory, so that repeated accesses to the same information can be handled without additional disk transfers. Because of locality in file access patterns, file caching reduces disk I/O substantially: typical systems today achieve 80-90% read hit rates with file caches of .5-5 Mbytes. File servers of the future are likely to have file caches with hundreds of Mbytes; this should be enough to hold all the files used by groups of dozens of users over periods of days or weeks (see [11] for supporting evidence). As a result, almost all read requests should be satisfied in the file cache.

Unfortunately, about 1/3 of file I/O (measured by either files accessed or bytes transferred) is writes [11], and most of this data must be written to disk. One possible approach is to place newly-written data in the cache and delay writes to disk for a while; some of the newly-written data will be deleted before they are written to disk, and disk I/O requirements will be reduced correspondingly (for example, [11] measured that 15-20% of all newly-written bytes are deleted within 30 seconds; in recent measurements of our current system, we found that 40% or more of all new bytes lived less than 30 seconds and 90% lived less than a day). However, the cache contents may be lost in a system crash or power failure. In order to provide reasonable assurances of file integrity, writes to disk probably cannot be delayed more than a few seconds or minutes. Thus the write traffic will limit the overall performance improvements from caching to about a factor of three or four.

Using the estimates of the previous section, caching should reduce disk requirements to 1-5 per user, assuming 100-MIP workstations. The low end of this range might be tolerable, but the high end would still result in the disk drives being the greatest part of a workstation's cost.

Although file caching appears unlikely to solve all the I/O problems by itself, it changes the nature of disk I/O in two ways. First, it shifts the balance of disk I/O from mostly reads to mostly writes. Almost all of the reads are satisfied by the cache, whereas most writes must eventually go to disk; the result is that disk transfers will consist primarily of writes. Second, caches provide a buffer for bursts of I/O (and I/O is notoriously bursty). Bursts of reads will mostly be satisfied in the cache; bursts of writes can be buffered in the cache, with the actual disk I/O performed asynchronously without requiring the writing process to wait.

At this point, two interesting issues emerge:

[1] Disk traffic in the future will become more and more dominated by writes, and most of the data written will never be read back (it will live in the file cache until deleted or overwritten). The main reason for performing disk I/O is as a safety precaution in case the cache contents are lost.

[2] Seeks limit disk performance. If files are only 3-4 kbytes long and two transfers are required per file, then only 3% of the raw disk bandwidth can actually be utilized. If files continue to be small on average, then the only hope for improving disk bandwidth is to reduce the number of seeks to much less than one per file.

Further improvements in file system performance may be obtained by addressing either of these issues. One general approach is to improve the reliability of the cache so that its contents can survive crashes and power failures. This would make it unnecessary to write data to disk until it is replaced in the cache. Since cache lifetimes will be much longer than average file lifetimes, almost all files would be deleted or overwritten before being replaced in the cache, so disk I/O rates would be reduced drastically. Sections 5 and 6 below suggest two possible ways for improving cache reliability. The second general approach is to change the disk organization so that many file accesses may be handled with a single seek. Section 7 introduces log-structured file systems as an approach along this line.

5. Solution #2: Caches with Battery Backup

One way to make file caches more reliable is to store them in memory with battery backup. This would allow the contents of the caches to survive power outages. However, the battery backup is only the first of several steps needed to ensure the reliability of cache data. In addition, the operating system would have to recover the contents of the file cache during reboot, and it would have to leave enough information around during normal operation to make this possible.

Even so, battery backup would not be sufficient by itself: the file cache would have to be able to survive operating system crashes as well as power outages. We hypothesize that most system crashes are of the ‘‘fail-stop’’ kind, where little or no damage has been done to the system before it halts itself for a reboot. If this is so, then it should be possible to build a file system that can recover the contents of the file cache during reboot. Unfortunately, there is very little data available on the failure modes of general-purpose operating systems; without such data, we cannot be confident that cache recovery is possible.

Another problem with battery backup is reconfigurability. Suppose a file server suffers a hardware failure requiring weeks to repair. Even assuming that the battery backup could last that long, it might be unacceptable for the data in the file cache to be unavailable during such a long repair period. Ideally, it should be possible to plug the cache memory card(s) into a different CPU and continue operation, just as disks can be moved in an emergency in today's systems. This suggests that the best organization for a battery-backup cache might be as a detachable ‘‘silicon disk’’. If the silicon disk were accessed like a very fast I/O device instead of like memory, then it would also be less likely to be corrupted during a system crash. If RAMs become sufficiently cheap and capacious, perhaps silicon disks will replace magnetic disks altogether.

If file caches become so reliable that information need not be written through to disk, then there would be very little disk traffic left: almost all information would live and die in the cache. The primary role of the disk would be as an archival storage area for infrequently-accessed data. In such an environment, we wonder whether current disk organizations are still appropriate. Perhaps it would make more sense to organize the disk structures around their archival function, e.g. with better versioning and enough redundancy to eliminate the tape backups required in today's systems.

6. Solution #3: Cache Logging

Another way to improve the reliability of file caches is to use a disk or array to hold a backup copy of cached blocks. As blocks in the cache are modified, their new contents could be written out to the backup disk in a sequential stream. The disk could be written in full-cylinder units, with only track-to-track seeks except for one long seek after each complete pass over the disk. Thus the disk would operate at nearly its full bandwidth. The cache could be reconstructed from the log disk after crashes and power failures.

Cache logging is the simplest of the solutions described in this paper, and could probably be added to existing systems without much difficulty. It would provide the same benefits as battery-backed-up cache memory, without the difficulties of ensuring cache integrity after crashes. Nonetheless, cache logging seems wasteful. If combined with a traditional file system structure, it would result in three levels of ‘‘permanent’’ storage: the cache backup disk, the real disk, and the tape backup. Information would rarely be read from any of these three levels; is it really necessary to have all three? In addition, cache logging would probably leave the main file storage disks nearly idle; if disk performance is a problem then it would be better if the combined bandwidth of all the disks in the system could be fully utilized.

7. Solution #4: A Log-Structured File System

Our most exotic proposal is to merge the cache log with the main file disks. We call this a *log-structured file system*, because the file system's representation on disk would consist of nothing but a log. As files are modified, both file data and header information are appended to the log in a sequential stream, without seeks. As with the cache-logging approach described above, this allows the file system to utilize the full bandwidth of a disk array, even if individual files are small (they can be collected into large blocks before being written to the log). Unlike the cache-logging approach, though, it uses only a single representation of information on disk.

In addition to their potential for improved disk bandwidth utilization, log-structured file systems have several other attractive properties:

Fast recovery. Log-structured file systems offer the possibility of much faster crash recovery than traditional random-access file systems. File systems that perform update-in-place, such as UNIX file systems, can leave the on-disk structures inconsistent if a power-fail or system crash occurs during a complex update. As part of rebooting the operating system must scan all of the disk maps and free lists in order to detect and repair these inconsistencies. Disk repair times can already be as high as a half hour or more for UNIX systems with a few Gigabytes of file storage; as file systems become larger, these times will increase. In contrast, all of the recently-changed information in a log-structured file system is at the head of the log; this results in cleaner failure modes. It should be possible to organize the log so that only a few blocks need to be examined near the head of the log to recover from crashes.

Temporal locality. In a log-structured file system, files that are written at about the same time will also be stored in about the same place on disk. This means that such groups of files can potentially be retrieved from disk with a single seek followed by one long read. If files written at the same time also tend to be read later in the same groups (which seems likely to us), then this will provide yet another boost in the performance of log-structured file systems. As a special case of this phenomenon, a large file written all at once will be stored perfectly contiguously on disk. This suggests that scientific applications will run well using a log-structured file system.

Versioning. The append-only nature of the log means that old versions of files are retained on disk. With a little extra bookkeeping effort, it should be possible to make these old versions available to users for recovery from certain kinds of user errors (e.g. an ‘‘undelete’’ command could be provided to recover from accidental deletions).

There are three difficult issues that must be resolved to make log-structured file systems practical. The subsections below describe the problems and some possible solutions. The first issue is how to handle the occasional retrievals that will be required from the log; Section 7.1 describes an approach that allows files to be retrieved from the log as quickly as today's best file systems. The second issue is how to manage log wrap-around; Section 7.2 discusses some possible solutions to this problem. The third issue is how to achieve efficient disk space utilization; this is discussed in Section 7.3. Finally, Section 7.4 summarizes the performance advantages and disadvantages of log-structured file systems.

7.1. Random Retrieval from a Log-Structured File System

Although we expect caches to reduce disk reads drastically, a log-structured file system must still provide a mechanism for retrieving information from the log. Most previous uses of logging require the log to be scanned sequentially to recover information from it. This would result in read performance thousands of times worse than today's file systems; even a very small number of reads could ruin the overall performance of such a system. Thus a random-access retrieval mechanism seems essential if a log-structured file system is to provide improved overall performance.

In a traditional file system such as 4.3 BSD UNIX [9] there are two steps in locating a file. First, its textual name must be translated into an internal identifier for the file. Second, given the internal identifier, the blocks of the file must be located. Name-to-identifier translation is accomplished by reading directory files (starting from a root or working directory whose identifier is known), so the whole problem boils down to locating the blocks of a file, given its identifier. Most systems accomplish this by using a map for each file, which describes where the file's blocks are located. For example, UNIX file systems place map information in a few fixed locations on disk. The internal identifier used to identify a file indicates where its map is located.

Unfortunately, the log approach doesn't permit maps in fixed locations, since this would require seeks to modify the maps whenever files are created, modified, or deleted. Instead, all new or modified information, whether data or map, must be written at the head of the log. Thus the key to providing random retrieval is to design a floating map structure that is integrated with the rest of the log. The paragraphs below describe one solution to this problem. It is not the only possible solution (see [3,4] for other examples), but it illustrates the feasibility of random retrieval in a log-structured file system.

Figure 1 illustrates in three steps how a traditional file system with a single fixed-location map can be turned into a log-structured file system with floating maps. Figure 1(a) shows a traditional disk, with a map array in one portion of the disk and file data in another. Figure 1(b) shows the first step towards a log-structured file system. New data blocks are now allocated sequentially from the head of the log; old data blocks are not re-used after deletion (but Section 7.2 below discusses how they might be re-used when the log wraps around). In Figure 1(b), however, the map array is still in a fixed location and must be updated in place whenever the file structure changes.

Figure 1(c) shows the second step towards a log-structured file system. The map array is now treated as a special file: the data blocks of this special file contain the maps. By treating the map array as a file, its blocks can float just like any other file: when a map entry is modified, its block of the map file is rewritten at the head of the log. An additional ‘‘super-map’’ gives the location of each block in the map

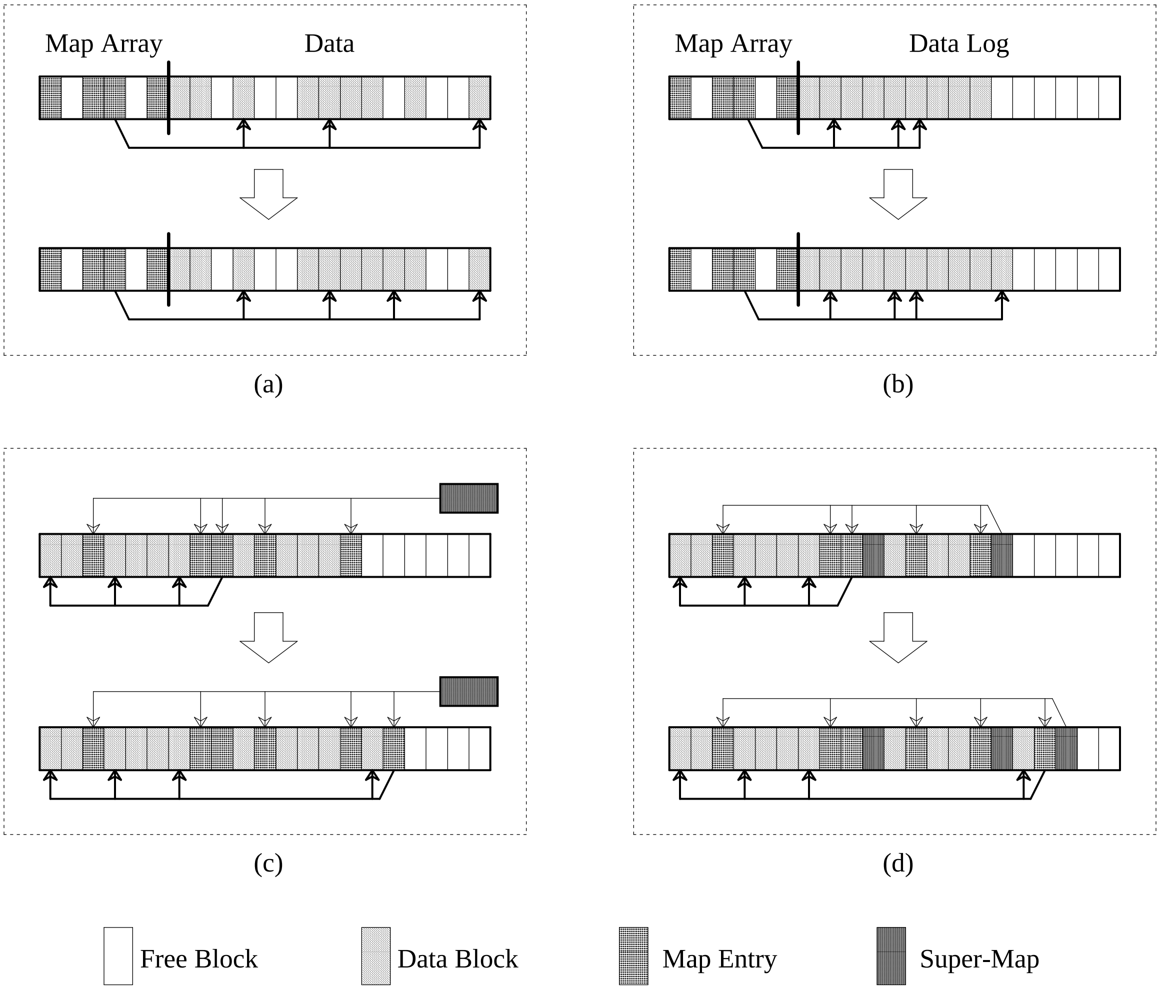

Figure 1. The derivation of a log-structured file system from a traditional one. Each figure shows the addition of a block to an existing file (''before'' is above and ''after'' is below). (a) shows a traditional file system with separate map and data areas; a new data block is allocated and the map is updated in place. In (b) the data area has been made into a log: each new data block gets added at the end of the log, but map entries are still updated in place. In (c) the whole disk is a log. The map array is treated like a file whose location is determined by a super-map; when a file is modified, a new copy of its map is appended to the head of the log and the super-map is modified to show the location of that map. In (d) the super-map is also written to disk to permit fast recovery after crashes.

file. Each reference to a map entry must consult the super-map to locate the entry on disk, but the super-map can be cached in memory just like all other file information, so no extra disk accesses are required.

The final step consists of occasionally writing the super-map to the head of the log. This is illustrated in Figure 1(d). The super-map should contain information that identifies it unambiguously as such. After a system crash, all that is needed to recover the file system is to search back through the log to the most recent copy of the super-map. The super-map should be written often enough that this is a short search. Then the log entries after the super-map can be reprocessed to regenerate the system state at the time of the crash. Writing the super-map to disk is something like an atomic file system commit: it encapsulates the state of the system at a particular point in time.

The culmination of all these steps is a file system with read performance nearly identical to our current file systems. At the same time, it makes possible the performance advantages that come from using a log approach to write data.

7.2. Log Wrap-Around

The second major issue in building a log-structured file system is how to handle wrap-around. No matter how large the log is, it will eventually fill up. By this time most of the information in the log will no longer be ‘‘alive’’: most of the files will have been deleted or overwritten. If a log takes hours or days to wrap around, most of the information in the log will be dead (according to our measurements, 90% will be dead if the log takes one day to wrap around). The issue, then, is how to recover the space that is no longer used while continuing to access the disk as a log.

The issues in dealing with log wrap-around are similar to the issues in garbage collection. One possibility is the stop-and-copy approach: stop the system and copy all the live information in the log to one end, leaving the free space all together at the other end. This approach seems infeasible to us because it would result in lengthy down-times for compaction, and because logs of practical size might wrap around in less than a day, so that compaction could not necessarily be carried out at night.

An alternative is to compact the log incrementally and continuously, so that the log is continually wrapping on itself. Before each new portion of log is written, the old information occupying the space must be examined. Dead information may be discarded, but live information must be saved by copying it, skipping it, or archiving it.

One possible approach is to compact the live information by copying it to the head of the log (see Figure 2). This approach will result in extra expense for long-lived files that get recopied every time the log wraps past them, even though they aren’t being used. A second approach is to leave the live information in place and add new information around it. This approach might result in extra overhead to skip over the live information, and it would cause the disk to become fragmented, so it is probably a bad idea except for very large blocks of live information. The third approach is to move some or all of the live information to another storage area. For example, there might be a hierarchy of logs where information gets archived from log to log as the logs wrap. This approach produces a result similar to garbage collectors based on generation scavenging [16]. It offers the possibility of integrating archival storage (such as optical disks or digital video-cassettes) into the system as the most senior level in the log hierarchy.

One of the most important issues in handling log wrap-around is how to identify the files that are still live. This must be accomplished with information in main

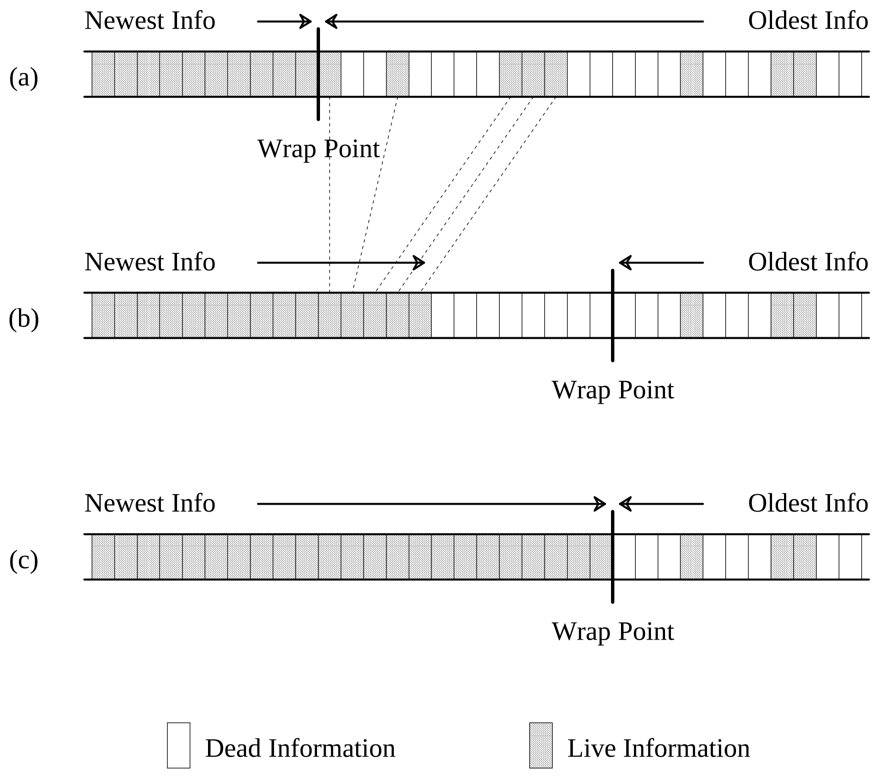

Figure 2. Incremental compaction in a log-structured file system. New data is appended to the log from left-to-right. In (a) the log has just filled up; among the oldest blocks in the log, only a few are still alive. In (b) the live information is compacted to the head of the log, leaving empty space for new log information. In (c) new information is appended to the log, regenerating the situation in (a).

memory or with information read back from the tail of the log; if seeks were required to retrieve information from disk, much of the performance advantage of logging would be lost. Since file maps are cached in main memory, the maps for most files will probably be available in memory at the time when liveness determination is made. If this turns out not to be the case, then some other data structure will have to be maintained in memory, such as a map of free and used log blocks, so that liveness can be determined without disk accesses.

Log wrapping introduces an overhead factor of two in disk bandwidth utilization: each log block must be read (to recover live information) before it is written. A disk seek might also be required to reposition the disk heads before writing. However the distance of such seeks will be small, and the log can be written in very large blocks, so the seek will be amortized over a large amount of data.

7.3. Disk Utilization

The third problem in implementing a log-structured file system is disk space utilization. Log wrap-around introduces a time-space tradeoff between the efficiency of disk bandwidth utilization and the efficiency of disk space utilization. Figure 3 illustrates the problem with some hypothetical scenarios. In general, higher disk space utilization implies that more information will still be alive when the log wraps around it, which means that more of the disk's bandwidth will be used to copy that information and less bandwidth will be available to write out new information.

Two factors determine the precise nature of the tradeoff. The first is the distribution of file lifetimes, which is more like Figure 3(a) than Figure 3(c). This factor is outside the control of the file system. The second factor is overall disk space utilization, which is under the file system's control. The file system can control space utilization by setting a threshold and refusing to allocate new files whenever the space utilization exceeds the threshold (for example, in 4.3 BSD UNIX, the threshold is 90%). This makes it possible to select any of a range of scenarios between the curves in Figure 3(a) and Figure 3(b). Table I shows the range of tradeoffs that will apply if file lifetimes are exponentially-distributed. For example, the second line in the table indicates that if users are willing to pay twice as much for disk storage (i.e., they only use half the available space), then they should be able to improve performance by a factor of 10 (40% bandwidth utilization instead of 3% in today's non-log-structured file systems).

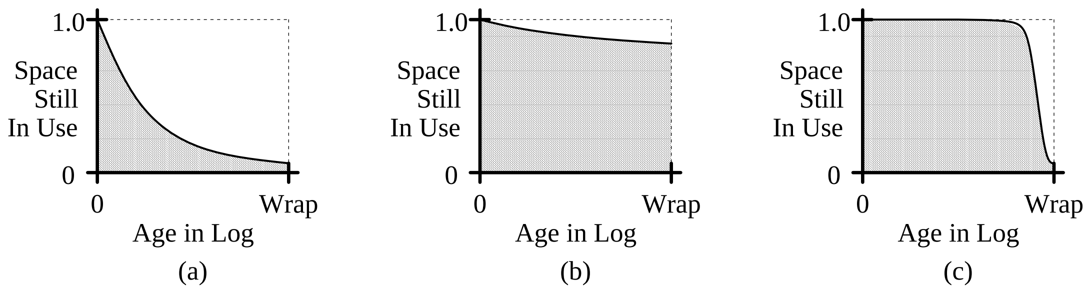

Figure 3. Some hypothetical scenarios for disk space utilization in a log-structured file system. In each figure, the x-axis corresponds to location in the log: **0** corresponds to the information just written, and **Wrap** corresponds to the oldest information in the log, which is about to be overwritten. The y-axis indicates the fraction of the information at that position in the log that is still in use: older information is more likely to have been deleted. The total disk space utilization is the ratio of the area under the curve to the area of the dotted rectangle. The work required during wrapping is related to the height of the curve at the **Wrap** point. In (a) most information has been deleted by the time the log wraps on it; this results in low overhead during log wrapping, but poor disk utilization. In (b) the disk space is almost fully utilized, but most of the disk bandwidth will be used to recopy information at the log head. (c) shows an ideal but unrealistic situation where files live until just before they are wrapped upon, then die at the last second so that they need not be dealt with at wrapping time.

Live Fraction	Bandwidth Utilization	Space Utilization
.1	.45	.39
.2	.40	.50
.3	.35	.58
.4	.30	.65
.5	.25	.72
.6	.20	.78
.7	.15	.84
.8	.10	.90
.9	.05	.95

Table I. The tradeoff between disk space utilization and disk bandwidth utilization. The first column lists the fraction of information still alive when the log wraps on it. The second column lists the fraction of raw disk bandwidth that can be utilized to write new information (this assumes that half the bandwidth is used to read back the old contents of the log; of the remaining half, some is used to rewrite live information and the rest is used to write new information). The third column contains disk space utilization, computed under the assumption of exponentially-distributed file lifetimes.

7.4. Performance Comparisons

In considering a log-structured approach, we were concerned primarily about small files that are read or written sequentially in their entirety and are usually present in the file cache. However, log-structured file systems also work well in a variety of other situations. Only one access pattern, files updated piecewise but read sequentially and frequently without locality, performs worse in log-structured file systems than in today's file systems.

By its very nature a log-structured file system will outperform other file systems for writes. In looking for weaknesses of the approach, it thus makes most sense to look at reads that miss in the file cache. For small files, a log-structured file system will have read performance at least as good as today's file systems (as shown in Section 7.1 above). In the worst case, one seek will be required for the file map and one for the file data. With a little cleverness in the log management, it should be possible to write the file map close to the file data so they can both be retrieved with a single seek. This would result in 2x better performance than current file systems.

For large-file reads, there are two cases to consider. The simplest case is files that are written all-at-once. These files will be contiguous in the log, which allows them to be read at least as efficiently as today's best file systems (particularly if the file map is written next to the data in the log). Random-access reads to such a file will require seeks, but no more in a log-structured file system than in a traditional file system.

The second case for large files consists of those that are written piece-wise, either by gradually appending to the files or by updating them in random-access mode. The logging approach permits such piece-wise writes and does not require the whole file to be rewritten, but the new data for the file will go at the end of the log. This will not be adjacent on disk to other data written to the file previously. If the file is later read sequentially from one end to the other, many seeks will be required. In comparison, a more traditional file system can keep the file's data contiguous on disk even under this sort of access pattern.

However, the *overall* performance for large piece-wise-updated files is not necessarily worse in a log-structured file system than a traditional file system. A log-structured file system performs the writes of the pieces with essentially zero seeks, but requires seeks for sequential reads. A traditional file system requires seeks for each of the writes, but can read back the whole file with one seek. If the file is written more often than it is read (or if the cache satisfies most read requests), then the log-structured file system will have better overall performance. Or, if the units of reading correspond to the units of writing, so that the file isn't actually read sequentially, then again the log-structured file system will have better performance (no seeks to write and one to read, vs. one to write and one to read). The only scenario where a log-structured file system is worse is for a file written in pieces, but read in larger blocks, which isn't accessed often enough to stay in the cache, yet has more read traffic than write traffic.

8. Related Work

The database community has used logging techniques for many years, both to reduce disk I/O during transaction processing and as a reliability and crash-recovery tool. However, in databases the log is a supplement to the random-access representation of the database, not a replacement for it as in a log-structured file system.

Logging approaches have begun to see some use in general-purpose file systems, but the motivation has usually been reliability or the limitations of write-once media. In contrast, our motivation for logging is to achieve high performance with read-write media. At least three recent projects have used logging as part of a file system for write-once media: Swallow [15], CDFS [4], and the Optical File Cabinet [3]. The Swallow system also included a generational approach to handle media with different characteristics, and the CDFS and the Optical File Cabinet papers describe mechanisms for random-access retrieval from their logs. These systems assumed infinite storage capacity or write-once media, so they did not address wrap-around issues.

Finlayson and Cheriton have recently implemented a system providing append-only log files and propose them as a general-purpose construct [2]. Finlayson and Cheriton's system allows retrieval from the log with cost proportional to the logarithm of the log size (in comparison to the constant cost of the mechanism in Section 7.1). However, Finlayson and Cheriton required all files to be append-only, rather than the more general UNIX-like model we have assumed, and they also assumed infinite storage capacity so they did not have to address log wrap-around issues.

One project where logging has been used to improve performance is Hagmann's work on the Cedar File System [5]. He used logs for file maps in order to provide high reliability of map information without performance degradation. However, Hagmann used logs only for the map information, so his mechanisms did not reduce seeks for file data. Furthermore, in Hagmann's work the log was a supplement to the normal on-disk structures, not a replacement for it.

9. Summary and Conclusions

If CPU speeds increase by a factor of 100-1000 between 1985 and 1995, then corresponding increases in I/O speeds must be found so that applications can scale smoothly in performance. We have described several possible solutions to this problem and think that log-structured file systems offer a particularly intriguing alternative for engineering and office applications. In comparison to today's file systems, they use up more of a cheap resource (disk bandwidth), but much less of an expensive resource (seeks). Log-structured file systems can make use of disk arrays to provide efficient writing, even for small files, and work well with main-memory file caches, which provide efficient reading. As a result, we think that overall I/O performance improvements of a factor of 1000 are feasible. Most of the improvement will come from having a hundred or so disks in an array; this provides the potential for a hundred-fold improvement in bandwidth or operations per second, if the operating system can keep all of the disks busy. Log-structured file systems will allow all of the disks to be utilized, and also provide about ten-times better bandwidth utilization of each disk than today's file systems.

Even if one of the other approaches, such as file caches with battery backup, proves better than log-structured file systems, we think that the nature of I/O to disk is changing enough to justify new disk organizations. Logging approaches offer easier recovery and versioning and better locality than most of today's file systems, so some of the logging techniques may be applicable as a supplement to other approaches.

Of course, the real proof is in the implementation. Section 7 pointed out several problems and possible solutions, but there are probably other problems that we haven't foreseen. The next step is to implement a prototype log-structured file system and analyze its behavior. We plan to do so, and encourage others to try similar techniques or different approaches to beating the I/O bottleneck, so that when 1000-MIP machines become available they will not spend all of their time waiting for I/O.

10. Acknowledgments

Garth Gibson, Doug Johnson, Mike Nelson, Randy Katz, Mike Stonebraker, and Herve Touati provided comments on early drafts of this paper, which substantially improved the presentation.

11. References

[1] Amdahl, G. ‘‘Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.’’ *Proc. AFIPS 1967 Spring Joint Computer Conference*, Atlantic City, N.J., April 1967.

[2] Finlayson, R., and Cheriton, D. ‘‘Log Files: An Extended File Service Exploiting Write-Once Storage.’’ *Proc. Eleventh Symposium on Operating Systems Principles*, November 1987, pp. 139-148.

[3] Gait, J. ‘‘The Optical File Cabinet: A Random-Access File System for Write-Once Optical Disks.’’ *IEEE Computer*, Vol. 21, No. 6, June 1988, pp. 11-22.

[4] Garfinkel, S., and Love, J. ‘‘A File System for Write-Once Media.’’ MIT Media Lab report, October 1986.

[5] Hagmann, R. ‘‘Reimplementing the Cedar File System Using Logging and Group Commit.’’ *Proc. Eleventh Symposium on Operating Systems Principles*, November 1987, pp. 155-162.

[6] Howard, J., et al. ‘‘Scale and Performance in a Distributed File System.’’ *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 51-81.

[7] Kim, M. ‘‘Synchronized Disk Interleaving.’’ *IEEE Transactions on Computers*, Vol. C-35, No. 11, November 1986, pp. 978-988.

[8] Lazowska, E., et al. ‘‘File Access Performance of Diskless Workstations.’’ *ACM Transactions on Computer Systems*, Vol. 4, No. 3, August 1986, pp. 238-268.

[9] McKusick, M., et al. ‘‘A Fast File System for UNIX.’’ *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.

[10] Nelson, M., Welch, B., and Ousterhout, J. ‘‘Caching in the Sprite Network File System.’’ *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 134-154.

[11] Ousterhout, J., et al. ‘‘A Trace-Driven Analysis of the UNIX 4.2 BSD File System.’’ *Proc. Tenth Symposium on Operating Systems Principles*, December 1985, pp. 15-24.

[12] Patterson, D., Gibson, G., and Katz, R. ‘‘A Case for Redundant Arrays of Inexpensive Disks (RAID).’’ *ACM SIGMOD 88*, Chicago, June 1988, pp. 109-116.

[13] Satyanarayanan, M. ‘‘A Study of File Sizes and Functional Lifetimes.’’ *Proc. Eighth Symposium on Operating Systems Principles*, December 1981, pp. 96-108.

[14] Schroeder, M., Gifford, D., and Needham, R. ‘‘A Caching File System for a Programmer’s Workstation.’’ *Proc. Tenth Symposium on Operating Systems Principles*, December 1985, pp. 25-34.

[15] Svobodova, L. ‘‘A Reliable-Object-Oriented Repository for a Distributed Computer System.’’ *Proc. Eighth Symposium on Operating Systems Principles*, December 1981, pp. 47-58.

[16] Ungar, D. ‘‘Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm.’’ *Proc. Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pp. 157-167.