September 1987 Report No. STAN-CS-87-1177

Log Files: An Extended File Service Exploiting Write-Once Storage

by

Ross S. Finlayson and David R. Cheriton

Department of Computer Science

Stanford University
Stanford, CA 94305

Log Files:
An Extended File Service Exploiting Write-Once Storage

Ross S. Finlayson
Stanford University

David R. Cheriton
Stanford University

Abstract

A log service provides efficient storage and retrieval of data that is written sequentially (append-only) and not subsequently modified. Application programs and subsystems use log services for recovery, to record security audit trails, and for performance monitoring. Ideally, a log service should accommodate very large, long-lived logs, and provide efficient retrieval and low space overhead.

In this paper, we describe the design and implementation of the **Clio** log service. Clio provides the abstraction of ***log*** files: readable, append-only files that are accessed in the same way as conventional files. The underlying storage medium is required only to be append-only; more general types of write access are not necessary. We show how log files can be implemented efficiently and robustly on top of such storage media-in particular, write-once optical disk.

In addition, we describe **a** general application software storage architecture that makes use of log files.

This work was supported in part by the Defense Advanced Research Projects Agency under **contracts N00039-84-C-0211 and N00039-86-K-0431,** by National Science Foundation grant DCR-83-52048, and by Digital Equipment Corporation, Bell-Northern Research and AT&T Information Systems.

1 Introduction

General-purpose logging facilities are necessary for computer systems and applications to meet requirements of reliability, security, and accountability. Using **a** logging facility, **a** subsystem or application can record, or *log,* its history of execution in sufficient detail to satisfy these requirements. Following a failure, the application can use this history to recover its current state, or to recover an earlier state. The history can also be used to restore the current state of a system after the data structures for this state have been revised, allowing the application to evolve without excessive disruption. This technique is often used to move between different incompatible versions of file systems, for example. Each of these uses of logging is based upon the principle that a system's true, permanent state is based upon its execution history, with the 'current state' being merely a cached summary of the effect of this history. We refer to this as the ***history-based*** model of system structuring.

In addition, a logged history can be examined to monitor for, and detect, unauthorized or suspicious activity patterns that might represent security **violat**ions.[1]

Standard magnetic disk-based file systems are inadequate for storing and accessing the large, **long**-lived logs that history-based applications may require. Because standard file systems maintain file access data structures in rewriteable permanent stor**age,** they cannot make effective use of more **cost**-effective, ***write-once*** storage media such as optical disk **[8].** By failing to distinguish (append-only) log files from other (rewriteable) files, standard file systems are unable to take full advantage of this new **tec**hnology.

In addition, standard file systems typically perform best on small files, with access to very large, continually growing files being more expensive. In particular, in extent-based file systems, such files use up many extents, since each addition to the file can end up allocating **a** new portion of the disk that is **discontiguous** with respect to the previous **extent.**[2] In indirect block file systems (such as Unix), blocks at the tail end of such files become increasingly expensive to read and write. (This is especially undesirable, because in many applications, the most frequent accesses to

[1]This assumes that the history itself cannot be circumvented or unduly compromised by such a violation.

[2]Since extent-based file systems tend to be faster for other types of files, factoring out these large, slow growing files eliminates or reduces one significant drawback of extent-based file systems.

large logs are to those entries that were written most recently.) In addition, the blocks of such files are likely to be scattered over the disk.

Furthermore, most file system backup procedures involve copying whole files, which is particularly inefficient (in terms of both time and space) for large log files, since only the tail end of the file will have changed since the last backup.

Finally, a log that becomes very large may have to be split over multiple physical disks. Most standard file systems do not support this capability.

In this paper, we describe the design and implementation of Clio,[3] a logging service that has been built for the V-System [4]. Clio provides the abstraction of log files-special readable, append-only files that are accessed in the same way as regular (rewrite able) files. Log files are implemented using write-once log devices (such as optical disk drives) attached to the file server machine.

The logging service is implemented as an extension of a conventional disk-based file server. It is able to use much of the existing mechanism of the file server, such as the buffer pool.

The next section describes how Clio provides efficient access to log files, as well as describing techniques for achieving reliability. In section 3, we analyze the performance of the system, in terms of both *access* time and space overhead. In section 4, we describe the history-based system structuring model in further detail. We argue that this model is well-suited to take advantage of the changing memory economics brought about both by the falling cost of RAM, and by the availability of increasingly high-density low-cost backing storage. Section 5 compares our approach to other work in this area.

2 Log File Service: Design and Implementation

A log *file* is a file of records (called log *entries*) that can be appended to indefinitely, and read back sequentially or randomly.

Log files appear the same as conventional file system files except that:

- log files are append only.

- when a log file is opened for reading, access can be provided to the sequence of entries in the file either subsequent to, or prior to, any previous point in time.

[3]In Greek mythology, "Clio" was the muse of history.

Otherwise, log files are named using the standard file directory mechanism, and are accessed and managed using the same I/O and utility routines that are used to access and manage conventional files [3].

As a special case, the entire sequence of log entries that have been written to a volume can also be considered a log file, called the *volume sequence log file.* The other log files are thus client-specified subsets (i.e. subsequences) of this sequence. These subsets are usually distinct, although the logging service allows a log entry to be a member of more than one log file.

Log files are implemented by an extended file server with one or more attached *log* devices. A log device is required to be a non-volatile, block-oriented storage device that supports random access for reading, and append-only write access. More general types of write access are not necessary. A *log volume is* the removable, physical storage medium, such as an optical disk, on which log data is stored.

Write-once optical disk drives are an attractive choice for the log device because they provide very high density storage (e.g. a 12" disk typically has a capacity of 1 Gbyte or more per side) at low cost (< 25 cents per Mbyte*) [2]. In addition, optical disks are very stable compared to magnetic disk,' and are removable, making them useful for archiving. Furthermore, the write-once restriction of current products is actually an advantage for a log device, because it improves the integrity of logged data. In fact, we favor a log device that is physically incapable of writing anywhere except at the end of the written portion of the volume.

The following key issues must be addressed when implementing log files:

Performance: How to provide efficient read access to the entries of log files, especially to entries that are located far back in time, without incurring excessive space overhead (over and above the space used to store client data).

Fault-tolerance: How to make the log service resilient to hardware and software failures, including failures that may result in random data being written to the log devices.

The following sections describe how we have addressed these issues in Clio.

*This compares with a cost of tens of dollars per Mbyte for magnetic disk.

This is true in part because the recording and reading heads of such devices can be further away from the storage medium than the heads of magnetic storage devices.

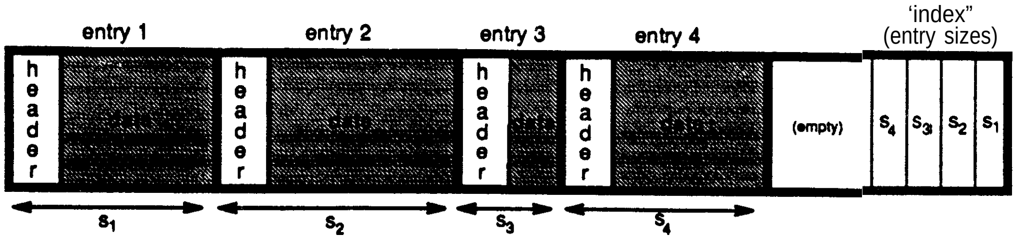

Figure 1: Layout of log entries within a disk block

2.1 Efficient Access

The key problem of efficient access is that of locating log entries that are far back in a log file, with low delay, and using few log device read operations. **Read** requests for *recent* data, on the other hand, are likely to be satisfied from the file server's in-memory cache. Write operation8 are performed only at the end of the written data-a disk location that is known at all times.

Each log entry record in a log file contains **a** *log* entry header, written by the server, that identifies the log entry. In particular, this header consists of the following fields:

local-logfile-id Identifies the log file(s) (usually only one) to which the log entry belongs.

timestamp The time at which the logging service received the written log entry. (optional)

size The size of the entry in bytes.

In practice, the **"size"** field **is** not stored in the header itself, but is instead stored in an index that is written at the end of each disk block, a8 illustrated in Figure **1.** This makes it easy to scan a disk block, either forward8 or backwards, to examine the log entries that it contains.

In principle, **a** log server could locate the **entries** that are member8 of **a** particular log file by **examining** every entry in every block of the volume sequence. This, of course, would be prohibitively expensive, especially if a desired entry is far away.

To efficiently **locate** the entries in log files, the server maintains a special log file called the ***entrymap*** log file. The **data** in this log file describes a sparse bitmap for each (other) log file, indicating which blocks on the log device contain log entries in this log file. In particular, the entries in the **entrymap** log file are as follows:

1. A 'level-l' **entrymap** log entry appear8 every N blocks on the log device. Such an entry contain8 a bitmap, of size N, for each active log **file**[6] that has entries in any of the previous N blocks. This bitmap indicate8 those blocks that contain such entries.

2. **A** 'level-2' **entrymap** log entry appears every N^2 block8 on the log device. Such an entry contains a bitmap, of size N, for each active log file that ha8 entries in any of the previous N^2 blocks. This bitmap indicate8 those group8 of N blocks that contain **suc** h entries.

3. and so on. . .

(The choice of N is discussed in section *3.)*

The entries in the **entrymap** log file effectively form a search tree of degree N, rooted at the ***k*th** level **entrymap** log entry, if ***k*** is the highest level **entrymap** log entry written. This **is** illustrated in Figure 2. In this example, where N = 4, we show only the bitmaps for one particular log file. The (five) blocks that contain entries in this log file are shaded. Having determined the block[7] in which a desired entry is located, the log server read8 **this** block and searches it sequentially for the desired entry.

Note that the **entrymap** log entries are located in well-known block8 (at regular intervals) on the log device, so the server is able to access them efficiently. Note also that the information in the **entrymap** log file is redundant, because, as mentioned earlier, it could also be obtained (although considerably less efficiently) by examining every log entry on the volume.

The server must also be able to efficiently locate the position of those log entries that were written at a given earlier point in time. The server uses a tree search, based on the timestamps in the log entry headers. **A** header timestamp is mandatory for the first log entry in each block, 80 the search **succeeds** to a resolution of at least a single block. At the upper level8 of the tree, the search uses those blocks that happen to contain **entrymap** log entries. Thus, when the server next attempts to locate the log entry (for a given log file) that is closest to the specified time, it

[6]**This does not include the volume sequence log file, and the entrymap log file itself.**

[7]**A log entry may also be fragmented over more than one block.**

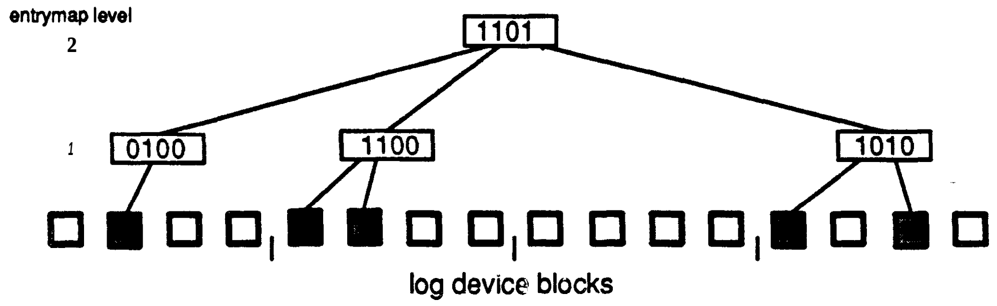

Figure 2: Example entrymap search tree for N = 4

is likely that the useful **entrymap** log entries already reside in the server'8 block cache.

Within a log file, a particular log entry can be uniquely identified using it8 timestamp. If the entry is written synchronously to the logging service, then a client can obtain this **timestamp** as a consequence of the write operation. This timestamp can subsequently be used to efficiently locate the log entry.

Some applications-for example, database transaction recovery mechanisms-need to uniquely identify a written log entry without the write operation being synchronous. **One** possible approach is for the client to use a unique identifier consisting of

1. a client-specified sequence number (that is written a8 part of the log entry), and
2. a client-generated timestamp.

To access the log entry, the timestamp is used to determine the approximate location of the entry within the log file. The sequence number is then used to identify the specific entry. The efficiency of this scheme depends on how well the client and server time clocks are synchronized. Its correctness depends on the sequence number not wrapping around within the maximum possible time skew between the client and the server.

The logging service allow8 a client to create **a** log file that is a ***sublog*** of an existing log file. If log file l_2 is a **sublog** of log file l_1, then any entry **that is** logged in l_2 will also belong to l_1. A client may **then** efficiently access either all of the entries in l_1, or **just those** in l_2. The **sublog** facility thus provides an additional way to efficiently locate a small, selected set of entries within a larger log file.

In addition, the **sublog** concept allows the familiar file naming hierarchy to be used in a natural way. For example, if "/" denote8 the volume sequence log file, and "/mail" denotes a log of mail messages delivered to a system, then "/mail/smith" may denote a log of mail **messages** delivered to user "smith". Note that each such name represents not only a log file, but also a directory of (zero or more) **sublogs.**

A log file may span several log volumes. Each log file is totally contained in one ***log*** *volume sequence-a* sequence of log volume8 totally ordered by the time of writing. Whenever a volume fills up, a (previously unused) successor volume is loaded, with this **successor** being logically a continuation of its predecessor. The newest volume in each volume sequence **is** assumed to be on-line, both for reading and writing. Many of the previous volume8 in a volume sequence may also be available for reading (only), or may be made available on demand, either automatically or manually. A log file can be uniquely identified by (i) a log volume sequence and (ii) a log file identifier relative to the log volume sequence. (This log file identifier is distinct from that of all other log file8 ever created on the same volume sequence.)

2.2 Reducing Space Overhead

Although Clio ha8 been designed to make use of **low-cost,** non-reusable storage media, we wish to avoid excessive storage **overhead** on the log device (that is, storage beyond that which is used for the client data itself). This is especially important if large numbers of relatively small log entries are written, for example, to support object-oriented databases.

We accomplish this goal primarily by limiting the **size** of log entry headers. The header for any given log entry **contains** information that is relevant only to **this** entry. Any information that is an attribute of a log file us ***a whole*** is recorded separately, in a separate log file called the *catalog log file.* Such "log file **specific**" attributes include a log file's name, its access permissions, and its time of creation. Any change to these attributes is also logged (at time of the change) in the catalog log file.

The simplest form of the log entry header is only 4 bytes in length, consisting of:

header-version: Indicates the form of log entry header that is being used. (4 bits)

local-logfile-id: Identifies the log file to which this entry belongs. This is an index into a table (called a catalog) of log file specific information (i.e. file descriptors) maintained by the server, and derived from the catalog log file. (12 bits)

size: The size of the entry. (16 bits)

Note that this information alone is sufficient to identify and parse every log entry in a block, as is necessary during server initialization, when the only information initially available to the server is the location of the last written block on the log device (and the location of the **entrymap** information).

With this minimal log entry header, the space overhead (due to the log entry header) for a log entry with d bytes of client data is $400/(d + 4)$ percent-for example, less than 10% for entries with more than 36 bytes of client data.

Entrymap log entries also contribute to the space overhead. An **entrymap** log entry, however, contains a bitmap for a log file only if this log file has entries that appear in the set of blocks covered by the bitmap. Therefore, log files that have few entries, or that are written to infrequently, incur little overhead in the **entrymap** log. On the other hand, although frequently written log files may be mentioned in numerous **entrymap** log entries, the overhead due to these **entrymap** entries is amortized over a large number of log file entries. In section 3.5 we show that under most circumstances the average overhead per log entry, due to the **entrymap** log entries, is small in comparison with that due to the log entry header.

2.3 Fault Tolerance

The log service must deal with the following faults:

File server crash The file server software or hardware fails, so the server has to be rebooted. The log service must be restored as soon as possible, -in spite of the loss of RAM memory contents.

Log volume corruption A software or hardware problem may cause garbage to be written to the log volume. Despite this, the log service must still be able to access any useful information that has already been written to the volume.

2.3.1 File Server Crash

If a file server crashes, we assume that the contents of its RAM memory are lost. On reboot, the log service, for each mounted volume, must reconstruct its cached knowledge of the log files that are maintained on this volume.

The server first locates the most recently written block on the volume. (If this block cannot be found by directly querying the device, then binary search is used.) The server then examines recently-written blocks, to reconstruct missing 'entrymap' information (that is, bitmap information for **entrymap** log entries that had still to be written at the time of the crash). Finally, the server reads the catalog log file, to determine which (client) log files are being maintained, and their attributes.

The major problem caused by a file server crash is the loss of data stored in volatile memory. For this reason, log entries are written synchronously to the log device when forced (such as on a transaction commit).

On a (purely) write-once log device, frequent forced writes **can** lead to considerable internal fragmentation, since a block, once written, cannot be rewritten to fill in additional contents. Ideally, in order to efficiently support frequent forced writes, the tail end of the log device is implemented as rewriteable non-volatile storage, such as battery backed-up RAM.

2.3.2 Log Volume Corruption

Log volume corruption must be assumed to occur, since a log volume may be written over a long period of time, during which hardware and software failures may occur. **A** failure may cause a portion of the log volume to be written with garbage. If previously written blocks have been corrupted, then the data in these blocks is assumed to be lost, unless the client(s) that wrote entries to the affected blocks have taken measures that allow them to recover from such an error. However, corruption of this sort should be rare, since the logging service attempts to enforce the append-only restriction at the lowest possible level of the system (ideally, in hardware). It is more likely that only previously unwritten blocks are corrupted.

In either case, the presence of corrupted blocks should not render the remainder of the volume unusable. For this reason, we do not simply abandon a corrupted volume and copy the log entries in the uncorrupted blocks to a fresh volume; this would be wasteful and time-consuming. Instead, corrupted blocks are invalidated (e.g. by overwriting them with all l's). The logging service ignores **a** block that has been invalidated in this way. If a previously unwritten block is corrupted, then its location is recorded in a special log file (thus allowing the server to locate this corrupted block if it is subsequently rebooted).

It is possible that an **entrymap** log entry is expected to occur in one of the invalidated blocks. If a block in which an **entrymap** log entry would normally be written has been invalidated, then the entry can be written instead in the next uncorrupted block, if such a block is nearby. In general, however, it is always possible for the logging service simply to assume that no such **entrymap** entry is present, at the cost of some additional searching of the lower levels of the **entrymap** search tree. This is true because the information in an **entrymap** log entry is redundant, and is present only to provide efficient. **access** to log entries.

3 Performance Analysis and Measurement

A production log service is expected to deal with volume sequences that are several hundred volumes long, containing millions of records, and running continuously for several years. Periodically, audit and monitoring processes read hundreds of records from various log **files** in the volume sequence. We are interested in the time and space performance of our design in this environment.

In this section we describe the measured performance of log writing and of log reading (given complete caching). We also analyze the expected performance of log reading in the absence of caching, and the expected space overhead, per log entry, due to **entrymap** information.

3.1 Log Server Configuration

Clio has been implemented in the V-System as an extension of an existing file server. The file server implements both regular file systems (i.e. with **rewrite**able files) and, using separate storage **devices,**[8] log file systems. Much of the code-in particular, the code that implements directory management and block **caching—is** common to the implementation of both **types of** file. The (Motorola 68000) binary image of the file server (alone) is roughly 90 kbytes in size; the addition of log file support has increased the size by less than 20%.

The current configuration uses magnetic disk to simulate write-once storage. Configurations with optical disk drives will be in use shortly. No change to the existing file server software is required in order to support optical disk configurations.

[8]or separate disk partitions

3.2 Log Writing

We measured the time taken for a client program to (synchronously) write a log entry to a log file. Both the client and the log server ran on a Sun-3 workstation. At the server end, the client data was copied to the server's in-memory block cache. The final write to the log device was performed **asychronously** with respect to the client; the cost of this operation is not reflected in these measurements. The server tagged each log entry with a complete, **14-byte** log **entry header** that included a (64bit) timestamp. **Entrymap** log entries were written 16 blocks apart (i.e. N = 16). The block size was I kbyte.

We measured both the time taken to write a 'null' log entry (i.e. a log entry containing no client **data**— just a timestamped log entry header), and the time taken to write a log entry containing 50 bytes of client data. The average time to write a 'null' log entry was 2.0 ms. For a 50-byte log entry, the average time was 2.9 ms. Of these times, 0.5 ms-1 ms were taken up by the basic synchronous client-server IPC (write) **operation.**[9] The cost of generating the timestamp was roughly 400 μs. The cost of 'maintaining and periodically logging **entrymap** information for this log file was low: only about 70 μs for each written log entry, on average.

We draw the following conclusions from these measurements:

- Excluding the cost of generating the header timestamp, the cost to the server of logging **a** small log entry in the block cache is **low**— comparable, in fact, to the cost of a basic local IPC operation.
- The cost (per log entry) of maintaining and logging **entrymap** information is generally negligible, especially if **entrymap** log entries are written infrequently.
- Attention should be paid to the cost of **generating** a timestamp for each log entry. We are **currently** investigating ways to more efficiently **access** kernel-maintained time in the V-System.

3.3 Log Reading

Reading a log entry **from** a given log file consists of three steps:

1. locating the block that contains the desired log entry

[9]The corresponding time for an IPC operation between different workstations is 2.5 ms–3 ms.

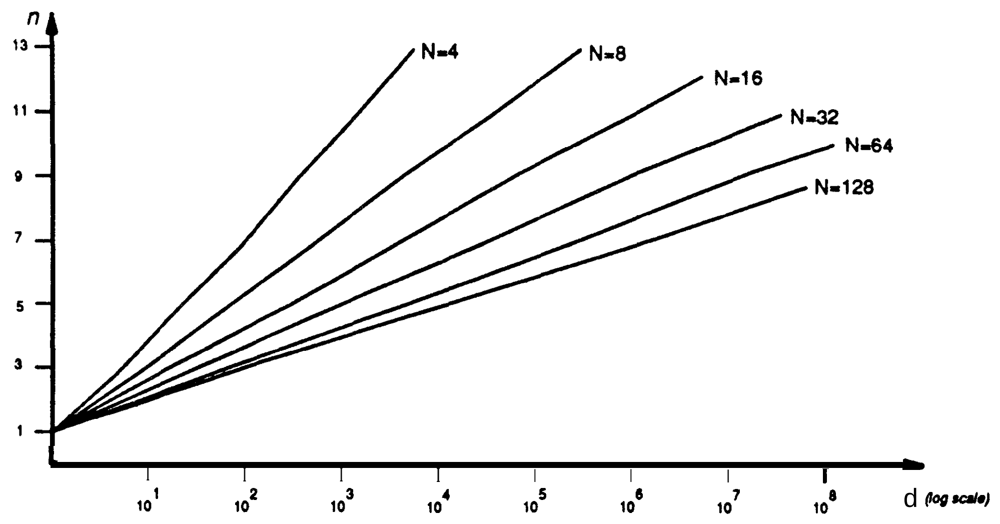

Figure 3: Theoretical average cost of locating an entry d blocks away (without caching)

2. reading this block (either from the server's block cache, or from the log device), and

3. locating and reading the desired log entry within this block.

The expected cost of steps 1 and 2 depends on how much of the necessary **entrymap** log information has been cached, and, in particular, on the way in which the log file entries are distributed on the volume. For example, the cache hit ratio for the blocks that contain the upper levels of the **entrymap** search tree will usually be quite high. We first discuss the worst case cost of step 1 (locating the desired block). This worst-case cost occurs if none of the necessary **entrymap** log information has been cached by the server.

3.3.1 Analyzed cost (no caching)

If the next (or previous) entry in a log file happens to be d blocks away from the current block (where we start looking), then it can be located by examining, on average, $n \approx 2 \log_N d + 1$ **entrymap** log entries, where N is the size of a bitmap in an **entrymap** log entry. Note that a block that contains a level-(i + 1) **entrymap** entry also contains a level-i log entry, so the number of actual *block* reads required may be even less than n.

Figure 3 shows n plotted against d (on a logarithmic scale), for various values of N. We are most interested in the performance of the system when d is large (that is, when the entry that we are trying to locate is a considerable distance away), since in such a case our 'no caching' assumption is most likely to be true. Also, for a storage device such as an optical disk, the seek time (which typically dominates the cost of reading a block) is greatest for blocks that are far away.

Note that for a given d, as N increases, n decreases by a factor of only about l/log N, so that there is little benefit in N being larger than 16 or 32, even for locating entries that are as many as 10^7 blocks away.[10]

Extensive log reading interferes with the performance of log writing, and vice versa. Thus, the log device should ideally have separate read and write heads. A separate write head makes it easy to physically enforce the rule that data is never rewritten, because the write head need only be able to move forward (plus be able to "restore" to the beginning of the disk when a new volume is mounted).

3.3.2 Measured cost (given complete caching)

Using the same client-server configuration as described in section 3.2, we measured the time taken for a client to read a 50-byte log entry from a log file.

A major component of this time is the time taken by the server to locate this entry using the **entrymap** search tree. This depends on the number of **entrymap** log entries that need to be examined, which in turn depends upon the distance (in blocks) between the log entry being read and the starting point of the search.

In Table 1, we list the measured cost of a log read, as the search distance is varied. In each case all disk blocks were located in (and therefore read from) the server's main-memory block cache. The search distance is listed as a power of the degree (N) of the

[10] Such a distance is not unrealistic given the likelihood of media with a capacity of 10 Gbytes or more.

search distance (blocks)	(if N = 16)	# of entrymap log entries read	# of disk blocks read (from memory)	time (ms)
0	(0)	0	1	1.46
N	(16)	1	3	2.71
N^2	(256)	3	5	3.82
N^3	(4K)	5	7	5.06
N^4	(64K)	7	9	6.51
N^5	(1M)	9	11	8.10

Table 1: Measured cost of a log entry read, for different search distances (given complete caching)

search tree. (We also indicate the actual distance, in blocks, if N is chosen to be 16.)

From these measurements we see that the cost of accessing (and interpreting, if necessary) a single cached disk block is around 0.6 ms. In comparison, a typical average seek time for an optical disk drive is ~150 ms [2]. Furthermore, queueing for disk reads (under conditions of heavy load) may make the average cost of a cache miss even higher. Therefore, the cost of a log read operation (which typically requires multiple block reads) is determined primarily by the number of cache misses.

When recently-written entries are read, both the **entrymap** information that is used and the log data itself will usually be in the cache, so the cache hit ratio should be very high. If, on the other hand, a log entry that is being read is located a large distance away, then neither the lower levels of the **entrymap** search tree nor the log data itself can be expected to be cached. A read of this type is expected to cost several hundred milliseconds. Fortunately, such reads are typically far less frequent than reads to log entries that are located nearby. If, for example, log entries within a log file are batched, so that each 'long distance' read is followed by a large number of 'short distance' reads, then the cost of each long distance read is amortized over the subsequent short distance reads.

3.4 Initialization

As was mentioned in section 2.3.1, server initialization consists of three main steps:

1. locating the most recently written block on the log device,
2. examining recently-written blocks, to reconstruct missing 'entrymap' information, and
3. reading the catalog log file, to reconstruct the log file specific attributes of each client log file.

Step 1 may require the use of binary search to locate the end of the written portion of the volume, at a cost of $\log_2 V$, where V is the size of the volume, in blocks.

The cost of step 3 depends on how many log entries need to be read from the catalog log file, and to the degree to which these entries are spread throughout the log device. Fortunately, much of the **entrymap** information that is used to locate these entries will already be cached at this point, as a result of step 2.

The cost of step 2 can be estimated as follows. To reconstruct missing level-1 **entrymap** information, the server need examine the blocks that were written since the last level-1 **entrymap** log entry was logged. There are between 0 and N such blocks (N/2 on average). Similarly, level-i **entrymap** information (for $i > 1$) can be reconstructed by examining between 0 and N recent level-(i- 1) **entrymap** log entries. In total, it may be necessary to examine N $\log_N b$ blocks, where b is the total number of blocks that have been written to the volume so far. On average, roughly $n = (N \log_N b)/2$ such blocks are read.

Figure 4 shows n plotted against b (once again, on a logarithmic scale), for various values of N. Note that this cost *increases* if N is increased. (This occurs because although a larger value of N increases the scope of **entrymap** log entries, it also increases the *separation* between them.)

This cost could be reduced by choosing a small value of N, by varying N so that it is smaller at higher levels, or by adding additional, redundant location information at higher levels (to reduce the separation between location information at these levels). However, the first two solutions have the drawback of increasing the cost of locating entries in the common case discussed earlier, while the third solution complicates the location algorithm somewhat, and also slightly increases the space overhead due to location information.

From this and the previous section, we see that a choice of N in the range 16-32 provides excellent performance for reading (even very sparse) log files,

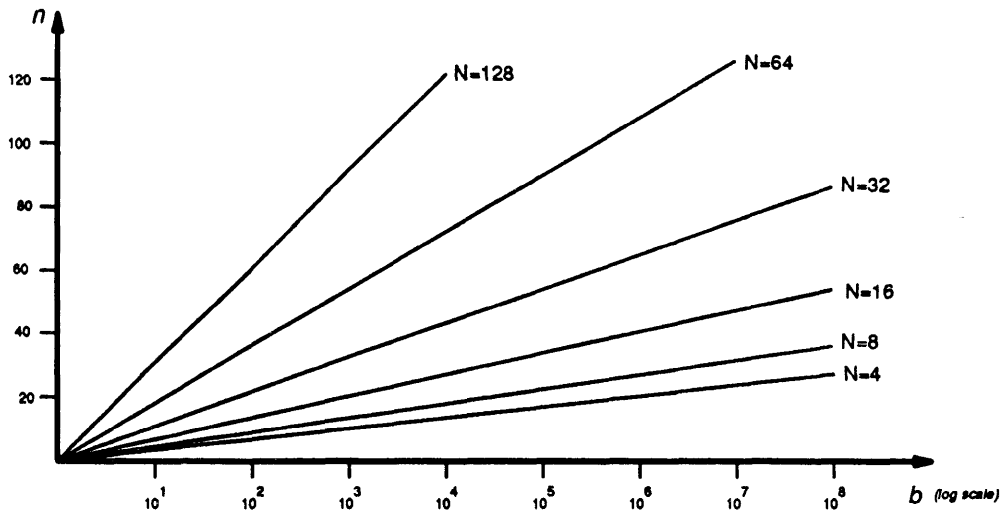

Figure 4: Theoretical average cost of reconstructing **entrymap** information

without leading to excessive overhead during server initialization.

3.5 Space Overhead

Section 2.2 described how space overhead is reduced. In this section, we sketch an analysis of the expected space overhead for our design, in terms of parameters that describe the size and distribution of written log entries.

The average space overhead, per log entry, is composed of

1. the average size, h, of a log entry header, and
2. the average per-entry overhead, o_e , due to the **entrymap** log entries.

o_e, in turn, is equal to $\epsilon(eE)$, where

e = the average number of **entrymap** log entries per block,

E = the average size of an **entrymap** log entry, and

ϵ = the fraction of each block that is taken up by the average log entry.

From the structure of an **entrymap** log entry, we see that $E = h + a(N/8 + c)$ (bytes), where a is the average number of log files referenced in an **entrymap** log entry, and c is a constant. Therefore, since $e < 1/(N-1)$, $o_e < \epsilon(h+a(N/8+c))/(N-1)$. Filling in likely values for many of these variables, namely: $h = 4$ bytes, $N = 16$ bits and $c = 2$ bytes, we see that o_e is (in this case) bounded by $0.27\epsilon(a+1)$ *bytes*. Thus, o_e is usually less than the overhead, *h, due to* the log entry header, unless the average log entry is large compared to a block (i.e. ϵ is large), and large numbers of different log files are typically written during short periods of time (i.e. a is large). For example, even if the average log entry were to take up an entire block, each block in a 16-block group would have to contain a different log file in order for the **entrymap** space overhead to be comparable to the header size.

We illustrate the space overhead that is incurred by an actual log file system, by considering a file system that we have been using to record user access (i.e. login/logout) to the V-System. Measured values of ϵ and a for this file system are roughly $1/15$ and 8, respectively. The average per-entry overhead due to **entrymap** log entries is therefore less than 0.16 bytes (which is less than 0.2% of the average entry size in this file system).

To summarize, most of the average per-entry space overhead is due to the average log entry header size, and this, in turn, can be kept low, as we showed in section 2.2.

4 History-Based Application Structuring

A *history-based* application has the following properties:

1. It uses an underlying (append-only) logging service for permanent storage, recording its *entire* persistent state in one or more log files.
2. The application's current state is an (at least partially) cached summary of the contents of these log files. This state can be completely reconstructed from the log files, if necessary.

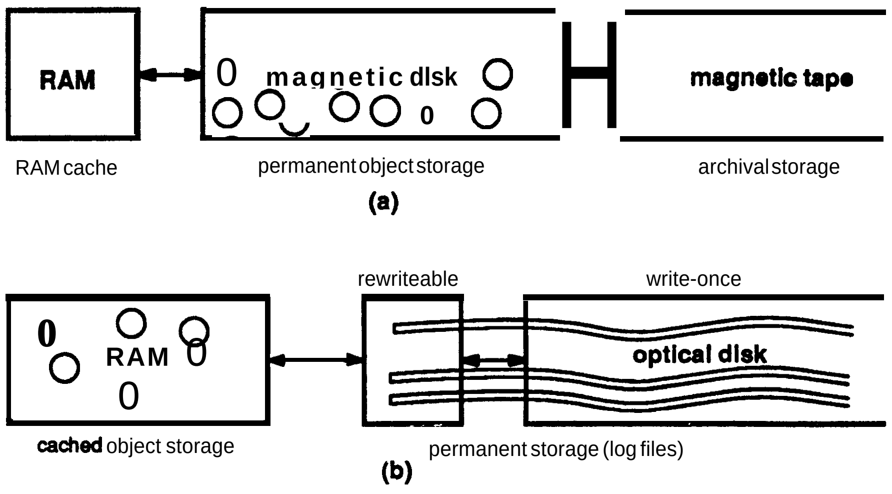

Figure 5: Storage models: (a) traditional. (b) history-based.

This model stands in contrast to the traditional model in which an application's persistent state is maintained in *rewriteable* permanent storage. These two models are illustrated in Figure 5. In the history-based model, the role of rewriteable permanent storage is less significant; it is used only as a staging area for the tail end of the log device, if at all.

The history-based model has a number of potential advantages over the traditional model. First, the history-based model combines regular permanent storage with archiving. No separate mechanism is needed for archival storage, which reduces the complexity of systems that follow this model. Similarly, because only a single form of persistent storage is used to restore the cached state of a history-based application, the recovery mechanism of such an application is simplified.

The append-only nature of this persistent storage also makes it easier to guarantee the consistency of the system state that is recovered from it. In particular, the logging service preserves the order that data is written to persistent storage, and ensures that if a log entry is recorded in persistent storage, then previously-written entries are also recorded. In addition, this model makes it possible to consistently access both a new (or tentative) version of an object, and a -previous version.

Furthermore, by using a logging service for permanent storage, a history-based application inherits many of the benefits of the logging service itself. In particular, this storage is less vulnerable to accidental corruption than it would be if it were rewriteable. (This is especially true if the logging service's append-only policy is physically enforced by the log device.)

Because a history-based application's current state is recoverable from the logging service, this state can be cached in RAM rather than on magnetic disk. The higher speed of RAM can make this cost effective, despite the fact that the cost of RAM, per byte, remains higher than that of disk. Suppose, for example, that the cost of retrieving 1 kilobyte is 100 ms if the data is read from a log device (on a cache miss), 30 ms if the data is read from a magnetic disk cache, and 1 ms if the data is read from a RAM cache. In this case, given the choice of adding R Mbytes of RAM versus D Mbytes of disk for the same cost, as long as the cache hit ratio for the RAM cache is at least 70% of the cache hit ratio of the disk cache, then the RAM cache has the better read access performance.

A potential disadvantage of the history-based approach is the possibility of excessive amounts of data being written to the logging service. Because this approach combines archiving with regular permanent storage, it is best suited for applications in which there is a close match between the time granularity that is sufficient for permanent storage, and that which is necessary for archival purposes, so that little of the logged data can be considered extraneous.

Two possible applications of the history-based model are illustrated below.

4.1 File Server Support

A conventional file service can be implemented following the history-based model. The file server maintains, in one or more log files, a file history for each file that it stores. The file history includes all updates to the contents and properties of files, as well as (possibly) information about read access to files. The file server can extract, from the file history, either the current version of a file, or an earlier version. (The contents of the current version are typically cached.)

When considering the feasibility of a history-based file server, two issues are of primary concern: (i) the size of the RAM cache that is needed to provide acceptable performance, and (ii) the rate at which log entries (representing updates to files) fill the log device. In each case, however, we are encouraged by measurements of the performance of typical file systems, such as the results obtained by Ousterhout [11]. In particular, Ousterhout's analysis of the Unix 4.2 BSD file system shows that cache miss ratios of less than 10% are possible with a cache size of only 16 Mbytes, given an effective cache write policy.

In addition, it was observed that typical file lifetimes are very short; for example, more than 50% of newly-written information is deleted within 5 minutes. This suggests that with an appropriate delayed write (or a "flush back") policy, most newly-written data will not lead to writes to the log device. Furthermore, temporary files can often be managed completely by clients, without any interaction with a network file server.

4.2 Electronic Mail

In a history-based mail system design, associated with each mailbox is a log file corresponding to mail messages that have been delivered to this mailbox. . The local mail agent maintains pointers into this "mail history". In addition, it caches copies of mail messages from the history, for efficiency. In this way, a user's mail messages are permanently accessible, and the storage of the mail messages themselves is decoupled from the mail system's directory management and query facilities, which can evolve over time without rendering old mail inaccessible. The Walnut mail system [7] was structured similarly, although it used rewriteable storage for logging, and allowed mail messages to be (permanently) deleted.

5 Related Work

Below we discuss the following classes of related work:

1. other general-purpose logging services that were designed to make use of write-once storage, and
2. other (actual or proposed) uses of write-once storage.

5.1 Other Logging Services

Daniels et al. [6] describe a distributed logging facility that was designed for use in transaction processing. In their design, a log, or *intervals* of consecutive entries within a log, may be replicated on a number of log server nodes for reliability, In our design, however, the basic service provided by a logging facility is that of a single, complete log file, with server-level replication (if any) being managed at a higher level.[11] In their design, the log servers and file servers are physically separate (at the cost of additional network overhead), rather than both services being integrated. Furthermore, log entries in their design are tagged with a *sequence number* rather than a timestamp. The timestamp in our design not only uniquely identifies log entries, but also makes it possible for them to be located by time. Finally, their design uses a binary tree structure to locate log entries. The performance of this scheme is within a constant factor of ours (both schemes have logarithmic performance---asymptotically the best possible), but our scheme requires significantly fewer disk read operations, on average, to locate very distant log entries.

The Swallow system [14] was a design for a reliable, long-term data repository that could use write-once storage media. This design is similar to ours in many ways. The major difference, however, is that Swallow was designed explicitly to support sequences of *versions* of *objects*, whereas our system is intended to support sequences of *arbitrarily-specified client data.* Specifically, this distinction has the following consequences:

- In Swallow, each object version (which is analogous to a log entry in our system) is linked to the previously written version of the same object. This link is the only 'location' information that is written to permanent storage. The design of Swallow was based on the assumption that almost all accesses are to the most recently written version of an object. It is impossible to scan forwards through an object history, without reading every subsequent block on the storage device. On the other hand, a general-purpose logging service, such as ours, needs to efficiently support a wide variety of access patterns.
- Swallow does not ensure that versions of different objects are written to the repository in the order of arrival; such an ordering is guaranteed only for different versions of the *sume* object.
- The Swallow design did not attempt to limit space overhead on permanent storage, perhaps because it was felt that large numbers of very small object versions would not be written to such a system.

[11] Note that our design does not preclude the possibility of replication occurring at the *log* device level (that is, with mirrored disks).

- As a reliable storage system for data objects, Swallow was designed with a built-in atomic transaction mechanism, in which 'tentative' versions of a set of objects are converted atomically into commit ted versions. An explicit atomic transaction mechanism of this sort would not be suitable for a general logging service.

5.2 Other Uses of Write-Once Storage

The CDFS file server [9] was designed especially for write-once optical disk. This design, like ours, assumes an append-only model of storage. However, CDFS was designed to support 'files' (and, in particular, versions of files) in the traditional sense, rather than supporting logs. In this system, a file is usually rewritten entirely whenever it is modified, The authors do, however, describe an extension of CDFS to support "fragmented files", so that only the modified portion of a file need be rewritten each time. Even in this case, a 'map' that describes the entire file contents must also be written along with the modified portion of the file. Thus, even with this extension, a large, constantly growing file could not be maintained without incurring excessive space overhead. Also, CDFS does not allow files to extend over more than one volume. Note that a general file system, such as CDFS, that has been designed to use append-only storage, could be implemented *on top* of our logging service (although with some loss of efficiency), by using a log file as its storage device. This would allow the same (physical) device to be shared with other applications.

The Amoeba file server [10] was also designed to make use of write-once storage, although some (non-volatile) rewriteable storage is also required. Newly modified pages of file data can be stored on write-once media, but higher-level **data** structures that describe 'file versions must be maintained in rewriteable storage.

Several authors (e.g. Rathmann [12] and Vitter [15]) have considered the problem of maintaining dynamic data structures on write-once media. This work has concentrated on the issue of how to create and update more complex data structures than logs-in particular, B-trees. In some cases, this work has assumed an append-only model of data storage, in which case these data structures could be supported on top of the log abstraction that we providing. Much of this work, however, has been based upon the (questionable) assumption that the write-once storage device allows arbitrary-sized, empty fields of a previously written block to be overwritten. In any case, it is our belief that frequent, fine-grain updating of write-once storage media to support dynamic data structures is a poor use of such media. It is more efficient for such data structures to be cached and updated in RAM, with the slower, write-once storage being updated less frequently, for checkpointing and archiving. In this way, we exploit the write-once nature of the storage medium, preserving the temporal ordering of updates, rather than attempting to hide it.

In [5], Copeland presents a number of arguments in favor of the "no deletion" (and "append-only") model of storage, and suggests that storage media such as optical disk make such a model feasible.

Spurred in part by the prospect of very high density permanent data storage, there has recently been increased interest in *temporal databuses,* that could support historical or time-dependent relations, and queries about past states of the database. Snodgrass and Ahn [13], for example, discuss the possibilities of databases of this type. In addition, a few authors (e.g. Ariav [1]) have proposed temporal data models (in most cases based upon the familiar relational data model) that attempt to provide a-semantic framework for such databases. Our work, however, is not aimed at attempting to provide the same facilities as **a** general-purpose database. Instead, we provide a simple data type-the "log file"-that is a natural extension of the familiar operating systems concepts of files and directories. Log files, although restrictive when compared to the more general data models, are easily implemented, and are useful for the many applications that require no more than simple logging. In addition, since log files provide an abstraction of append-only storage media (hiding block and volume boundaries), they can be used to support any data type that could be implemented directly on top of suc h media.

6 Conclusions

We have described the Clio logging service--an extension of a conventional logging service that provides access to Jog *files:* readable, append-only files that are accessed in the same way as conventional files. We have shown how the logging service is able to make effective use of write-once storage. In addition, we have illustrated how applications may make use of such a service.

We draw the following conclusions from this work. First, log files can be implemented on write-once storage devices such as optical disk, allowing efficient retrieval, and low space overhead. We have provided some insight into the time-space trade-off that arises when trying to provide fast read access to log files.

In addition, we observe that log files fit naturally into the abstraction provided by conventional file systems, since such files can be accessed in the same way as regular append-only files. A uniform I/O interface, such as the interface [3] used in the V-System, supports access to this type of file. Our experience in incorporating the log file implementation as part of an existing file server has been favorable. The combined implementation allows for the sharing not only of hardware resources, but also of code. It also provides the file server with particularly efficient access to log files. (This is important, since we plan to implement atomic update of (regular) files, using log files for recovery.)

Finally, we observe that the append-only storage model is appropriate even if the backing storage medium happens to be rewriteable. An append-only storage policy helps guard agains data corruption, as well as providing accountability and built-in archiving.

Our work to date--designing and implementing a general-purpose logging service-represents the first step in exploring history-based storage management. We plan to explore this approach further by developing applications that follow the "history-based" paradigm, as well as extending our file service in this manner. We are working to gain greater experience with such applications, and with Clio itself.

7 Acknowledgements

Peter Brundrett implemented and maintained significant portions of the V-System file server, on which Clio has been built. Our interest in the history-based structuring model was inspired by discussions with Jim Donahue. Bruce Hitson provided helpful comments that improved the presentation of the paper.

References

[1] G. Ariav. A temporally oriented data model. *ACM Transactions on Database Systems,* 11(4) :499–527, December 1986.

[2] A. E. Bell. Optical data storage-a status report. *Proceedings of the 6th IEEE Symposium on Mass Storage Systems, 93-98,* 1984.

[3] D. R. Cheriton. UIO: a uniform I/O system interface for distributed systems. *ACM Transactions on Computer Systems,* 5(1), February 1987.

[4] D. R. Cheriton. The V kernel: a software base for distributed systems. IEEE *Software,* 1(2), April 1984.

[5] G. Copeland. What if mass storage were free? *Computer, 27-35,* July 1982.

[6] D. S. Daniels, A. Z. Spector, and D. S. Thompson. *Distributed Logging for Transaction Processing.* Technical Report CMU-CS-86-106, Carnegie-Mellon University, Department of Computer Science, June 1986.

[7] J. Donahue and W-S. Orr. *Walnut: Storing Electronic Mail in a Database.* Technical Report CSL-85-9, Xerox Palo Alto Research Center, November 1985.

[8] L. Fujitani. Laser optical disk: the coming revolution in on-line storage. *Communications of the* ACM, 27(6):546–554, June 1984.

[9] S. L. Garfinkel. A file system for write-once media. October 1986. MIT Media Lab.

[10] S. Mu1lender and A. Tanenbaum. A distributed file service based on optimistic concurrency control. *Proceedings of the ACM Symposium on Operating System Principles,* 51-62, December 1985.

[11] J. K. Ousterhout et al. A trace-driven analysis of the Unix *4.2* BSD *file* system. *Proceedings of the ACM Symposium on Operating System Principles,* 15-24, December 1985.

[12] P. Rathmann. Dynamic data structures on optical disks. *Proceedings of the IEEE Data Engineering Conference,* 175-180, April 1984.

[13] R. Snodgrass and I. Ahn. Temporal databases. *Computer,* 19(9):35–42, September 1986.

[14] L. Svobodova. A reliable object-oriented data repository for a distributed computer system. *Proceedings* of *the ACM Symposium on Operat*ing *System Principles,* 47-58, December 1981.

[15] J. F. Vitter. An efficient I/O interface for optical disks. *ACM Transactions on Database Systems,* 10(2):129–162, June 1985.