HEWLETT PACKARD

Ivy: a study on replicating data for performance improvement

Sai-Lai Lo

Hewlett-Packard Laboratories
Concurrent Computing Department

HPL–CSP–90–48
14 December 1990

Keeping multiple copies of a file may improve read performance by increasing the bandwidth through parallel access; by balancing the load across storage nodes and by choosing the one which gives the fastest response. However, keeping all the copies up-to-date could incur a significant overhead. One possible approach to achieve good read performance and avoid slow updates is to change the number of copies according to the usage pattern. The aim of the Ivy project was to assess the potential of this dynamic replication scheme in achieving better read performance. A series of trace-driven simulations were performed. The results of this study indicate that read performance improvement of over twenty percent can be acheived with dynamic replications.

Ivy: A study on replicating data for performance improvement

Sai-Lai Lo

December 14, 1990

Abstract

Keeping multiple copies of a file may improve read performance by increasing the bandwidth through parallel access; by balancing the load across storage nodes and by choosing the one which gives the fastest response. However, keeping all the copies up-to-date could incur a significant overhead. One possible approach to achieve good read performance and avoid slow updates is to change the number of copies according to the usage pattern. The aim of the *Ivy* project was to assess the potential of this dynamic replication scheme in achieving better read performance. A series of trace-driven simulations were performed. The results of this study indicate that read performance improvement of over twenty percent can be acheived with dynamic replications.

1 Introduction

In a file system, keeping multiple copies of a file may improve read performance. The transfer time may be shortened due to the ability to transfer data in parallel from a number of copies simultaneously. The load on various storage nodes could be evened out because there is more than one choice where a request can go to. This in turn shortens the queueing time of requests and hence reduces the turnaround time. Moreover, under certain conditions, such as when there are more than one copy on a single disk or when it is possible to query all storage nodes which hold at least one copy of the file, the time spent on disk head movement, defined as the seek time, can be reduced by choosing the closest copy.

The study of keeping multiple copies of files, known as file replication, in a distributed system has been an active area of research in the past decade. Various algorithms have been developed and have been applied in some early distributed file system designs[Sov84]. More recently, with increasing concern on the availability of data, several projects, such as Coda[SKK89] of CMU and Echo[MHS89] of DEC SRC, are investigating the design of replicated file servers. Their emphases are on improving the availability of data and minimizing the overhead usually associated with most replication algorithms. Although many designs have provisions to reduce the overhead in reading from a replicated file, little has been done to study how replication can speed up access to frequently-read files.

Moreover, we do not know of any study on linking the degree of replication with the actual usage pattern. In many distributed file systems, the degree of replication is defined on a per volume basis. Each volume is typically a subtree of the complete file system and each contains a large number of files. For example, both Andrew[MSC+86] and Athena[Tre88] have replicated read-only volumes. Multiple copies of files are kept on different file servers and copies are normally read-only in that they can only be updated by system administrators. More recent designs, such as Coda and Echo, have replicated read-write volumes. Although the two designs differ in various aspects, such as consistency guarantee under failure conditions, they are similar in that all files within a replicated volume are replicated to the same level. LOCUS[WPE+83], designed in early 80's, allows users to choose which files within a volume should be replicated but the degree of replication, i.e. how many copies should be kept, is again defined on a per volume basis. Saguaro[Pur87] provides greater flexibility in defining the degree of replication of files. However the flexibility has been achieved by requiring the users or agents of users to specify explicitly how many copies are to be kept and where to store them.

Although keeping more copies of files might in principle improve read performance, the increase in overhead on updates could offset this benefit. In many replication algorithms, such as weighted voting algorithm[Gif79], transactions are needed to propagate updates in order to guarantee consistency. Some algorithms which do not need transaction support require a master site to coordinate updates to objects. Besides, updating replicas would increase the load on the disks. Clearly, the need to update all copies in synchrony imposes a restriction on the number of copies we would otherwise like to keep. Therefore we might consider devising an algorithm that automatically grows the number of replica for frequently-read objects, and is able to cut down on the number of copies before updates are performed. We shall refer to this approach as *dynamic replication* throughout this report.

The highly parallel architecture of DataMesh[Wil89] enables us to explore the possibility of using dynamic replication in a far more aggressive way to achieve better read performance. In a DataMesh, each storage node consists of a disk bundled with a multi-mips CPU and several megabytes of memory. There could be over a hundred of such nodes and they are interconnected by a very high speed network. The interconnects between storage nodes, which are physically at most a few feet away could be far simpler and faster than a local area network which typically is used today to connect file servers. Since the communication protocol between storage nodes could be far simpler than is needed in a local area network, a remote memory access which allows a node to access the memory of another node could be two orders of magnitude faster than a typical remote procedure call. Therefore it is possible to design the storage system with very close interactions among storage nodes. For instance, before a read request is dispatched, it is possible to choose the node which will give the fastest response by querying all nodes that contain copies of the object. Such queries are impractical in conventional file server designs. Also, copies of files could be made or discarded quickly when the need arise. Given these unique characteristics of a DataMesh architecture, it is therefore appropriate to investigate the use of dynamic replication in improving read performance of storage systems.

This project is a study to find out what potential, if any, a dynamic replication scheme has in achieving better storage systems read performance.

This project is not a design of the algorithm needed to perform such task.

We want to know

1. In deciding the degree of replication,
   - what observable parameters are important and
   - what heuristics should be used.
2. And most importantly, to quantify the performance improvement through various means we suggested at the beginning of this section.

In the next section, our project approach is discussed and is followed by two sections detailing various aspects of our study. The results obtained are analyzed and presented in section 5 and 6. Our report then ends with our conclusion.

2 Approach

2.1 Trace-driven simulation

We performed a series of trace-driven simulations to study the effect of various dynamic replication strategies. Trace-driven simulation involves the gathering of data from working systems and the transformation of these raw data into suitable form for use as input to the simulator. This could be quite involved but was nevertheless needed because it is very difficult to synthesize credible workload to suit our purpose. Since we want to experiment with changing the degree of replication according to the files usage pattern, a realistic pattern is essential to our study.

2.2 Workload characteristics

We took traces from two HP9000 series 800 machines. One of them is a departmental time-sharing system usually with about eight users log-on. It is used for document preparation, news reading, electronic mail, and departmental information storage. It also serves as a backup server for the workstations in the department. The other system was mainly used by one user for program development. The systems were usually lightly loaded. We expect dynamic replication to have greater effect in performance on more heavily loaded machines as there are more opportunities where keeping multiple copies would help, such as in balancing the load over a number of disks.

There is always a question on whether the workloads we studied are representative. We took the view that these are real workloads and many file usage characteristics such as the ratio of reads to writes, lifetime, size etc. are very much related to what the system is used for and varies little across different systems used for the same purpose. However, we expect systems used for other purposes such as CAD/CAM or database etc. to have different file usage characteristics. Hence our results should be treated as a first step in understanding the problem and should be repeated with traces collected on other systems used for other purposes.

2.3 Multiple copies on multiple spindles

We only considered keeping multiple copies of whole files on different disks. Nevertheless, with rotating disk storage technology, we might gain performance if multiple copies are kept at different locations on a disk because the chance is higher to find a copy close to the current disk head position. We choose not to study this for two reasons. Firstly, the file and disk layout, which determine where blocks of a file are stored, would play a very important role in determining the effectiveness of multiple copies and deserves a separate investigation. Secondly, multiple copies on a single disk could only have the effect of reducing seek time and doesn't have any effect on load balancing and reduction in transfer time. With the constraint of the duration of the project, this possibility was left for future investigation.

2.4 System call level vs physical IO level trace

Our simulator accepted input at the system call level rather than the physical IO level. Hence, in addition to the disk IO path, the characteristics of the HP-UX file system, which is almost the same as the BSD4.2 file system, have to be simulated as well. Simulation at this level is needed because much of the files usage information would be lost if traces are collected at the physical level.

On the otherhand, simulation at the physical IO level is far simplier as these characteristics have already been filtered out. However at this level, much of the file system usage information has been lost. It is very difficult to distinguish disk traffic due to virtual memory paging, file system meta and real data IO. It is not possible to relate which file a disk block belongs to. And hence it is not possible to tell when a file is deleted and when it is read or modified.

2.5 Simulating HP-UX file system

Simulation at the system call level with attention to details down to physical disk IO level is quite involved. Two points are noted here. First, not every read or write system call would result in a physical disk IO. Second, a physical disk IO related to a system call may occur before, in case of read ahead, or after, in case of delay write, the dispatch of the call. Our interest is on the effect of keeping multiple copies on different spindles. Hence our experiment should be a controlled one in which every aspect which could affect the overall performance should be as close to the real system as possible while only the number of copies and locations of copies are permitted to change. To achieve this, we either measure the effects of various system aspects from the working system or simulate it in our simulator. The following is a list of factors that our experiment took into account.

Buffer cache A large percentage of file read system calls are satisfied by the buffer cache. Similarly, a number of file write system calls are buffered in the cache and either the data is written to disk asynchronously (the system call returns before the write has completed) or the write to disk is deferred until the buffer is reused or the data has stayed in the buffer longer than a predefined time. We monitored and recorded the buffer cache read hits and had to simulate the asynchronous and delayed writes.

File layout Since we are interested in the reduction in seek distance with dynamic replications, rather than assuming a simple random function for seek distance, the actual locations of disk blocks corresponding to each reads and writes were monitored and recorded. And in our subsequent simulations, we assumed that all copies are stored at the same location on all disks. With this assumption, all replicas are laid out on disks in exactly the same way and there is no bias in favor of any copy which could happen if there is a difference in file layout among the replicas.

Read-ahead HP-UX file system implements a read-ahead policy to detect sequential file access and speed up the turnaround time. When a read hits a disk block boundary, a disk IO is scheduled to read in the next block but the IO is not associated to the process. This policy was implemented in our simulator.

3 Trace gathering

3.1 Raw trace data collection

Our traces were collected on HP9000 series 800 machines running HP-UX version 7.0. The HP-UX built-in measurement system provides a set of useful kernel instrumentation points as well as tools and drivers for collecting these information in user programs. Adding new instrumentation points to the kernel is easy. There is a set of kernel measurement routines which packs and timestamps data and put them into a kernel buffer. The flexibility in adding new instrumentation points proved to be very useful. As will be seen later, our project demanded new data from the kernel which are not available from the instrumentation points that come with release 7.0.

Interface to the measurement system is via a utility program which periodically scans the kernel buffer and copies the data to its standard output channel. Individual instrumentation points can be turn on and off at anytime via a utility program, without the need to recompile or reboot the kernel. Since the buffer space for storing measurement data in the kernel is limited, when the system is busy, the buffer could wrap around before the utility program can catch up with the generation rate. This would result in the loss of trace data. The problem was avoided by increasing the kernel buffer to a large size (512Kbytes) and making the utility program read the buffer once every two seconds. In addition, we allocated a separate disk to store our trace data temporarily so as not to interfere with normal disk traffic.

The following built-in instrumentation points were used:

- bread_read
- breada_read
- breada_reada
- bwrite
- bdwrite

- exec
- exec_after_args
- exec_fname
- exit
- fork
- syscallexit
- syscallstart

A detail description of these instrumentation points can be found in [Cha90]. These were not sufficient for our purpose in two ways. Firstly, there was no way to assign a unique identifier to a file. Secondly, the information about file deletion was missing. Therefore we added two additional instrumentation point and they are described below:

create_open2 The format of the record is described in table 1. When a file is opened or created, a record of this type is emitted only if the file in question is an ordinary file in a local file system. In other words, no record would be emitted for NFS file systems, pipes, character devices etc. The distinction is necessary to filter out non-file-system related IO. However to make the instrumentation point useful for other purposes, we should probably emit records of this type for all open/create calls and add a type field to the record to identify what types the files are. A unique identifier(UID) can be formed by combining fsid, inid, and geid. The uniqueness is guaranteed by the monotonically increasing generation number (geid) which is incremented when an inode is freed and reused.

unlink The format of the record is described in table 1. A record of this type is emitted for every unlink system call to delete an ordinary file in a local file system. However the actual file might still exist after the call because the link count has not reach zero, as the Unix semantics demand. This can happen, for instance, if the file is opened by some processes and the file is deleted only when the file is closed subsequently. Unfortunately the structure of the kernel code makes it very difficult to report the status of the file with respect to the link count. As a consequence of this, our simulator has to treat the unlink information as advisory only and have to be ready to correct any inconsistency if later information reveals so.

3.2 Trace data filtering

One major problem in capturing the dynamic behavior of systems is that the amount of data involved is so large that it is impractical to store them all. For instance, on a normal working day, a trace gathered on a time-sharing system with about eight users logged on and collected over a period of 12 hours occupied around 200 MBytes of space. These included records of all system calls and buffer cache activities. Since we were to collect data continuously for several weeks, some work had to be done to extract useful information from the raw trace before they were stored on stable media.

To associate a logical segment of a file to a list of disk block addresses and to record whether the data needed for a read are in the buffer cache, we needed information from both the system call and the buffer cache. With a read system call, a **syscallstart** record is followed by a sequence of **bread** records and ends with a **syscallexit** record. We can associate the read/write system call with the block cache reads/writes during the processing of the call because each block cache record is tagged with a **pid** and at anytime, only one system call is in progress for each process. One complication is that every buffer cache access would result in a record being emitted and these accesses may not be for data blocks but are for reading and writing index blocks and cylinder group block free-list etc. To find out which block records were data block references, the kernel source was studied to understand the sequence of block cache accesses during the processing of a read/write file system

Trace point	Field	Size (bytes)	Description
create_open2			
	pid	4	process id
	fsid	4	file system identifier
	noid	4	inode number
	geid	4	generation number
	size	4	file size
unlink			
	pid	4	process id
	fsid	4	file system identifier
	noid	4	inode number
	geid	4	generation number
	size	4	file size
	mtime	8	last modified time
	atime	8	last access time
	ctime	8	last changed time

Table 1: Data format of new trace points

call. Since this filtering process depends on intimate knowledge of how the coding is done, it is very HP-UX implementation dependent.

Having found out the data blocks associated with a read request, cache hit can be easily identified as each block cache record has a hit field which tells whether the block requested is actually in the buffer cache.

In Unix, every opened files are referenced by a file number, for example, 0 is the standard input, 1 is the standard output and 2 is the standard error. In every read or write system call, the file is referenced using the file number. This number, together with the **pid** would allow the kernel to locate the inode associated with the file after going through several index tables. Since these tables are hidden inside the kernel, we have to recreate some of these states in order to translate the **pid** and file number pair to the UID of the file. This was done by keeping track of all the files opened and associated each file number of every process with the UID of the corresponding file. However attentions had to be given to system calls like **fork** with which the child process would inherit all files opened by the parent and other calls like **dup** which duplicates a file handle. Similarly the current position of the file index which is stored inside the kernel has to be recreated somehow and attentions had been given to all system calls that could affect the file index position such as **lseek**, **truncate** and **open**. Moreover, when both the parent and the child processes share the same file handle, the change in the file index position has to be reflected in both processes. For example a shell might fork a child to execute parts of the shell script that the parent would continue from the point left behind by the child.

In our first filtering program implementation, every read and write were emitted as discrete records. This created enormous data files, about 400Mbytes in 24 hours. On closer examination, part of the reason was that many reads/writes were in tens of bytes chunks. For instance, a syserr log daemon was invoked every ten minutes. This daemon used a library routine to scan the kernel image symbol table and read the kernel image file 20 bytes at a time. Since a large percentage of read calls were satisfied by data in the buffer cache, we changed to emit read records only if at least one data block required by the read was not in cache. To preserve all the read information, we added to the close record the number of system calls invoked since the file was opened, the total amount in bytes read/written and whether all read calls were satisfied with data in the buffer cache. While not emitting all read records did amount to loss of data but the information retained was enough for our purpose. With this compaction, the data size was reduced by 10 times to 20-40Mbytes daily, which was manageable.

We implemented a filter/converter program which accepted raw data via pipe from the measurement system's utility program and emitted the filtered and compacted data via the standard output which could then be directed to other programs, such as *compress*, or stored directly to disk. We ran the filter/converter program in parallel with trace gathering and even with our program running simultaneously, there was no perceivable performance degradation but this approach could be less satisfactory on heavily loaded machines or ones that are low on real memory.

We only stored our trace data on a local disk of the measured system for at most a day. Every midnight a cron process was dispatched which stopped the trace gathering program and started a new one. The previous day's trace was then shipped via the network to a 20G optical juke box for long term storage. To restart our trace gathering process when it crashed accidentally or killed by our cron process, a shell script was written which would start a new one 30 seconds after the last one crashed and a log file was kept to record all these events. With these provisions, the trace gathering process was pretty automatic and needed little attention during the period of tracing.

The program captured the data reasonably well except that about 0.2% of the reads could not be translated. Closer examination revealed that this was because when a sync daemon flushes dirty buffers out, the buffer cache trace records are tagged with the pid of the current process, and if this happens to be the process which is doing a read/write system call, these extraneous records would mess up our routine to sort out the data block address and hence the whole record has to be rejected.

3.3 Trace data format

The following is a description of the types and the formats of the records in the trace data emitted by the filter/converter program. The type definition of the records are in C syntax. Moreover, the following data type definitions are used to describe the data formats of the records.

```
typedef struct {
    unsigned tv_sec;
    unsigned tv_usec
} TIMEVAL;

typedef struct {
    unsigned fsid : 16;
    unsigned geid : 16;
} FS_GE_ID

typedef struct {
    unsigned block:27;  /* block number */
    unsigned size:4;    /* size of block (in Kbytes) */
    unsigned hit:1;     /* 1=hit, 0=miss */
} BLK_REC
```

3.3.1 tr_exec

```
typedef struct E {
    TIMEVAL timeval;
    char    pathname[];
} tr_exec
```

The record contains the timestamp of the exec system call and the full pathname of the program executed.

3.3.2 tr_create_open

```
typedef struct {
    TIMEVAL   timeval;
    unsigned  noid;
    FS_GE_ID  fs_geid;
    unsigned  size;
} tr_create_open
```

Record every create/open system call on regular files only. the tuple [*noid,fs_geid*] uniquely identifies a file. The size of the file is returned in *size* and is zero if the file is newly created.

3.3.3 tr_read

```
typedef struct {
    TIMEVAL   timeval;
    unsigned  servicetime;
    unsigned  noid;
    FS_GE_ID  fs_geid;
    unsigned  start;
    unsigned  length;
    BLK_REC   blkrec[];
} tr_read
```

Record every read system call **with at least one block cache miss.** *Timeval* is the time the system call started. *Servicetime* is the time taken, in micro seconds, to process the call. *Start* gives the logical offset in bytes from the beginning of the file. *Length* is the number of bytes requested. *Blkrec* is an array of block records indicating where this chunk resides. No record is emitted for reads which are satisfied totally by the data in the block cache. Additional information is recorded in *tr_close.*

3.3.4 tr_write

```
typedef struct {
    TIMEVAL   timeval;
    unsigned  servicetime;
    unsigned  noid;
    FS_GE_ID  fs_geid;
    unsigned  start;
    unsigned  length;
    BLK_REC   blkrec[];
} tr_write
```

Record every write system call. See the description of tr_read.

3.3.5 tr_close

```
typedef struct {
    TIMEVAL   timeval;
    unsigned  noid;
    FS_GE_ID  fs_geid;
    int       total;
    int       rdwrcount;
} tr_close
```

Emitted when a file is closed. *Timeval* gives the timestamp. *Total* gives the number of bytes read/written since the file is opened/created. *Rdwrcount* gives the number of read/write syscall calls executed since the file is opened/created. If the sign of *total* is negative, all the system calls are read. Otherwise one or more of the system calls are writes. If the sign of *rdwrcount* is negative, all the read system calls are satisfied with data in the buffer cache. Otherwise there are some calls which have cache miss.

3.3.6 tr_unlink

```
typedef struct {
    TIMEVAL   timeval;
    unsigned  noid;
    FS_GE_ID  fs_geid;
    unsigned  size;
    TIMEVAL   mtime;
    TIMEVAL   atime;
    TIMEVAL   ctime;
} tr_unlink
```

Emitted when a file is unlinked. *Size* gives the size of the file when it is unlinked. *Mtime,atime* and *ctime* give the last modified time, last access time and last changed time of the file respectively. It should be noted that a file could be unlinked and not deleted either because the file is still opened or because the link count is not zero.

4 Simulator

Our simulator modeled the queuing and servicing of requests at the disk interface. Each disk is a service center which can handle only one request at a time. While a request is being processed, newly arrived requests will be queued. We are interested in the queuing time, service time, turnaround time and seek distance distributions when subject to input from our traces and under different configurations.

The simulator can be divided into five functional blocks. These are: *Disk*, *Copy selection*, *Dynamic replication control*, *Buffer cache* and *Event arrival adjustment*. In the following, the assumptions behind the design of our simulator are presented, this is followed by a description of the functional blocks of the simulator and the section is ended with some notes on the implementation.

4.1 Assumptions

Before moving on, we would like to list the assumptions and simplifications we made in our simulator:

1. We did not model other system resources, such as memory, process cycle, I/O channel etc., and hence not the contention for these resources. Part of their effects were reflected in our trace and since in a DataMesh environment, the processor power and I/O channel bandwidth are plentiful and the processing time and delays caused by these resources will be insignificant when compared to the time elapsed in getting data in and out of the disk.

2. In a DataMesh architecture, there are *spigot nodes* which receive requests from the network. For each request, the spigot node first locates the master node that contains the copy and then forwards the request to the node. The master node in turn have to locate all the copies available and collects information from each node containing the copy before the request is dispatched to the most appropriate node. In addition, during the updates of replicated data, there would be a number of message exchanges between replicas. Our view is that these exchanges between nodes must be performed in a time period significantly less than the disk

service time and hence were ignored in our study for simplicity and to obtain an upper bound on the performance benefits attainable.

3. Although we recognized that the disk traffic needed to create a new replica is significant in our model, we did not simulate this sort of traffic to simplify our problem because of a lack of time. Instead we assumed that the creation of replica was done in the background and a new replica would appear magically sometime after the decision to create a new copy was made. We foresee that in a real implementation such traffic would be put on a queue separated from real requests and will be given lower priority. Also the spare IO capacity should be sufficient to handle such traffic.

4.2 Disk

The disk functional module simulated the characteristics of a disk. There are several aspects of a disk which demanded attentions:

4.2.1 seek time

This is the time to move the disk arm to the desired track. The following two formulae express the seek time as a function of the distance in number of cylinders the disk arm has to move. The formula for a HP7937 disk drive is:

$$seektime = \begin{cases} 0 & ,\mathrm{d} = 0 \\ 4.119 + 0.890\sqrt{d} - 0.004d & 0 < \mathrm{d} \leq 384 \\ 12.909 + 0.18d & ,\mathrm{d} > 384 \end{cases}$$

The formula for a HP7935 disk drive is:

$$seektime = \begin{cases} 0 & ,\mathrm{d} = 0 \\ 3.993 + 1.676\sqrt{d} - 0.025d & ,\, 0 < \mathrm{d} \leq 342 \\ 19.502 + 0.21d & ,\mathrm{d} > 342 \end{cases}$$

Each formula was obtained by curve-fitting a plot of the IO time (seek time + rotational latency + transfer time) vs seek distance (number of cylinders) for 80,000 eight Kbytes transfers. With these formulae, we were able to accurately estimate the seek time, given the disk block address we gathered in the traces.

4.2.2 rotational latency

This is the time for the requested data to rotate under the head. The average latency to the desired information is halfway around the disk. For a disk rotating at 3600 RPM, the average rotational latency is 8.3ms. We assumed a random distribution for rotational latency. This, however, did not reflect the actual situation as HP-UX file system could be fine tuned to interleave disk blocks to match the processing speed of the rest of the system. During sequential access to files spanning multiple disk blocks, the rotational latency could be quite small since the processing of the next request overlaps with the time for the desired information to appear under the disk head. Hence during sequential access to large files, our simulator would tend to overestimate the rotational latency.

4.2.3 transfer time

This is the time to transfer a block of bits under the disk head. It is a function of the block size, rotation speed, recording density of a track, and speed of the electronics connecting disk to computer. The transfer rates for the disks we used are 1Mbytes per second.

4.2.4 controller overhead

This is the time imposed by the disk controlling hardware in performing an I/O access. The overhead is 3.5ms for HP7935 and 1ms for HP7937.

4.2.5 disk scheduling algorithm

When there are more than one request waiting to be served, the order in which these requests are processed is determined by the disk scheduling algorithm. In HP-UX, the CSCAN (cycle scan) algorithm is used. However because of the lack of time to refine our implementation, we used simple FIFO scheduling instead.

4.3 Copy Selection

For a file with multiple copies stored on multiple disks, this functional module decides which disk a read request should be dispatched to. It selects the disk that will provide the shortest overall turnaround time. In our simulator, we computed what the service time (seek time + rotational latency + transfer time + controller overhead) would be for each request before it is put on one of the disk queues. Since we are using FIFO scheduling, the queuing time for a new request is simply the sum of the service times of all requests already on the queue. The sum of the queuing time and the would-be seek time for each copy is calculated and the copy with the lowest value is chosen.

It should be noted that if CSCAN disk scheduling is used instead of the simple FIFO scheduling, the arrival of a new request could cause a reordering of the queue and we would have to recalculate the service time for all the requests on the queue.

4.4 Dynamic replication control

With dynamic replication, there are several parameters which could affect the overall file system performance. Two of these are:

Degree of replication Number of copies of files kept in the system.

Placement of copies Where the copies are kept.

We shall refer to the algorithm selecting the values of these two parameters as a replication policy. Our study is to find out what is the best heuristic in maximizing read performance improvement due to replication and minimize the overhead of updates to these copies. This functional module implements the replication policy and can be easily modified to experiment with different algorithms.

4.4.1 Degree of replication

The decision of how many copies of a file are kept is based on the past usage pattern. Therefore the reference history of files are kept in the simulator. The information kept include the following:

1. File size
2. Creation time
3. Last read time
4. Last write time
5. Number of times a file is opened for write
6. Number of times a file is opened for read
7. Number of times a file is opened for read and have to fetch data from disk

Our replication policy defines the number of copies as a function of these information. Exactly how these information are used would vary between different replication strategies. For instance, one strategy might specify that more copies would be made if a file exists for longer than 5 minutes and is read more than 5 times. Another one might specify that files with size larger than 10 Mbytes will not be replicated at all.

The decision to change the degree of replication is triggered by some events which change the status of a file. The events include the following:

1. File create
2. File open for write
3. File open for read
4. File truncation
5. File reads
6. File writes

Depending on the replication policy, a file truncation event or a file open-for-write event might trigger a drastic cutback in the number of copies kept in the system. On the otherhand, a file open-for-read event might trigger a decision to increase the number of copies of the file.

In addition, the replication policy has to decide in what steps the number of copies of a file be changed. It could be a gradual increase and decrease, within certain upper and lower bounds, or a step change to a fixed number.

Once the new number of copies of a file is decided, there is a choice as to when the newer copies are to spring into existence. It could be immediately or it could be deferred.

4.4.2 Placement of copies

Another part of the replication policy is the placement of copies. When there are more storage nodes than the number of copies, a choice has to be made as to where the copies are distributed among the nodes. The placement of copies is important because a poor distribution would skew the load on a few storage nodes and hence negate the benefits of load-balancing among storage nodes holding copies of files.

Also, when the number of copies of a file is reduced, a choice has to be made to choose which copies are to be removed. If only one copy is to be kept, the retained copy could be the one where the file is first created or could be chosen randomly among the copies for better load balancing.

4.5 Buffer cache

As was discussed in the previous section, there are certain aspects of the Unix buffer cache which could not be measured and recorded in our trace. These aspects, however, are important to perform realistic trace-driven simulation. Hence, the purpose of this functional module is to simulate these asynchronous behaviors of the file system involving the buffer cache.

4.5.1 Delay and asynchronous write

The HP-UX file system writes data to buffer cache in three ways:

Synchronous write Schedule a write to disk and block waiting for the IO to complete. This is mainly used by the file system to write updates to inodes and directories as well as other file system meta data. This can optionally be specified by the users but is rarely used.

Delay write Put the data into the buffer and set the appropriate flags in the buffer header to ensure that the data is written out before the buffer is reused. The call then returns. A synchronization daemon which runs every 30 seconds flushes all modified buffers which have stayed in the cache long enough.

Asynchronous write When the end of a cached block (normally 8 Kbytes in size) is written, the system assumes that the block will not be accessed in the future and a disk IO is schedule immediately to write the data to disk. The call then returns without waiting for the IO to complete.

We assumed that all writes are either delay writes or asynchronous writes. In our simulator, all delay writes are queued and are combined if two or more writes are directed to the same block. Either these writes are dispatched as asynchronous writes when the end of the blocks are written or when they have been on the queue for more than 30 seconds. In addition, delay writes are not dispatched if the files which originated these writes are deleted.

4.5.2 Read ahead

The HP-UX file system executes an algorithm which try to anticipate the need for a second disk block when a process reads a file sequentially. The second IO is scheduled asynchronously in the hope that the data will be in memory when needed, hence improving performance.

To detect sequential read, the HP-UX kernel saves the next logical block number and during the next iteration, compares the current logical block number to the value previously saved. If they are equal, the physical block number for read ahead is calculated and an asynchronous read for the second block is scheduled if the first and the second blocks are not in cache.

The simulator fully implemented this behavior. Each file can at anytime have two reads in progress.

4.6 Event arrival adjustment

In most event-driven simulations, request arrivals are assumed to be independent events. This assumption is not totally valid in trace-driven simulations of file systems. When a file is accessed, it is usually done via a sequence of reads or writes system calls. Since, at any time, there can only be one outstanding system call for each Unix process, it follows that the sequence of reads or writes are temporally ordered. Therefore, while the requests generated by different processes can be considered as independent events, the requests from the same process are not. We could imagine that there is a *think time* between two requests originated from the same process. This *think time* is a characteristic of the user and should be independent of how our simulated systems behave.

This functional module adjust the request arrival time to keep the user *think time* constant. Ideally this should be adjusted on per process basis but since we did not have process information in our trace, we did this on per file basis. This should work most of the time since concurrent reads to a file are very rare. Even if this does happen, it is likely that many of the reads to the same region of the file would have found data in the buffer cache already.

4.7 Implementation

The simulator was based on a DataMesh simulator written by Stephen Brobst (another student). The idea was to augment the functionality of the DataMesh simulator to accept traces obtained from working systems. However, due to a lack of time, this goal was only partly accomplished and a trim-downed version of the simulator was adapted for our use.

5 Traces characteristics

This section describes the systems and the trace data used to drive the simulations of dynamic replication policies. To provide better understanding of the characteristics of the systems studied, for every file appearing in the trace, we extracted the following information:

- creation time
- last read time
- last write time
- deletion time
- size
- number of times a file was opened read-only
- number of times a file was opened read-only and required reading from disk to fetch data.
- number of times a file was opened read/write or write-only

To facilitate the understanding of our data, in some of the graphs below, we further separate our data into four categories:

1. Files which were created before the trace and still existed at the end of the trace. Since our trace period lasted for several weeks, it is safe to consider that files belonging to this category were long term files.
2. Files which were created and deleted during the trace. This category includes files with lifetime ranges from several seconds to several weeks. This category contains both short term and medium term files.
3. Files which were created before the trace and were deleted during the trace.
4. Files which were created during the trace and still existed at the end of the trace.

5.1 Characteristics of the studied systems

Traces were collected from two computer systems. The characteristics of these systems are described below.

5.1.1 Machine "Cello"

- This is a departmental time-sharing system and usually has about eight users logged on.
- It is a HP9000/845 with 64 Mbytes of main memory, of which 10% is used as the file buffer cache.
- The system is running HP-UX version 7.0.
- There are nine partitions on six disks. Two of the partitions, one each on disk 0 and disk 1 respectively, are used as the swap space. The remaining seven partitions are mounted to the file system.
- In total, there are approximately 2.5 Gbytes of disk storage on-line.
- The system normally runs standalone with no remote file systems mounted.
- From midnight till early morning everyday, the system receives news feed and the group's workstations' backups through the network. Incremental backup to tape is done every night and a full backup is done every month. During normal working hours, the system is mainly used for document preparation, news reading, electronic mail etc.

- Table 2 lists the disks and partitions mounted. The root directory resides on disk 0. Disk 1 contains program source files. Disk 2 contains the temporary working directory and a file space for storing data files, such as system traces, too large to be stored in normal users' file space. The users' files are stored on disk 4. The news feed from the network are stored on disk 5. Also the system acts as a backup server for a number of workstations in the group and the backup data are stored on disk 6. Usually, the disks were kept at 60% to 90% full.
- The trace started at 00:00, Saturday, 1 September 1990 and ended at 00:00, Saturday, 29 September 1990.
- The size of the trace is approximately 700 Mbytes.

Device no.	Disk address	Disk type	Size (Kbytes)	File system mount point
13	0.13	HP7935	344148	/
267	1.11	HP7935	319630	/usr/local/src
519	2.7	HP7935	75348	/tmp
523	2.11	HP7935	319630	/mount/logs
1026	4.2	HP7937	558051	/users
1282	5.2	HP7935	394979	/backup
1538	6.2	HP7935	394979	/usr/spool/news

Table 2: Disk configuration of "Cello"

5.1.2 Machine "Triangle"

- This system was used throughout the tracing period by a single user doing program development.
- It is a HP9000/835 with 24 Mbytes of main memory, of which 10% is used as the file buffer cache.
- The system is running HP-UX version 7.0.
- There is a HP7937 (approx. 500Mbytes) attached to the system which contains the swap space and the root file system.
- The system normally runs as a stand-alone system with no remote file systems mounted.
- The trace started at 00:00, Friday, 7 September 1990 and ended at 00.00, Sunday 30 September 1990.
- The size of the trace is approximately 70 Mbytes.

5.2 File size distribution

Figure 1 to figure 8 show the file size distributions of the file systems studied. Each figure is a plot of the cumulative frequency against file size, that is, the y-axis shows the proportion of files with size smaller than or equal to the value shown on the x-axis. Log scale is used for the x-axis because of the large range of file sizes. The overall file size distribution on **Cello** is not plotted because the large number of news files would dominate the distribution.

Some interesting features of the graphs to note are:

- As expected, news files on **Cello /usr/spool/news** are comparatively smaller than the files on the other file systems. 80% of the files are smaller than 2Kbytes and less than 1% are larger than 20Kbytes.
- **Cello /users**, **Cello /usr/local/src** and **Cello /mount/logs** have similar file distributions. 70% of the files are smaller than 8 Kbytes. The similarity is expected because they are all used to store program sources, documents etc. However, there are some large files on **/mount/logs**, as this disk is used as a scratch pad area and short term storage, and some large data files such as system traces are stored there.
- **Cello /** is similar to the three file systems described above except that there is a jump in the graph between 700 and 900Kbytes. These files are probably news packets received from the network.
- Not all the features in these graphs are easily explainable, for example, we do not understand the jump in the graph of **Cello /tmp** at around 120-130Kbytes, or the apparent logarithmic distribution of file size on **Cello /backup** but we do expect the distribution of this file system to be different from the others because this is used to store backup files.

5.3 File system composition

Table 3 lists the number of files that appeared in the trace for each file system on **Cello**. Figure 9 shows these files divided into the four categories described at the beginning of this section. Figure 5.3 shows **Triangle /** divided into the four categories.

It should be noted that the figures do not represent the composition of the file systems at any moment in time. These are the accumulation of all files created and deleted over the trace period as well as files which already existed before the trace. Nevertheless, these diagrams do reflect the ways these file systems were used. **/usr/local/src** mainly consists of long term files which are program source. **/tmp** consists almost entirely of short and medium term files. **/backup** contains backups of the groups' workstations' and these backup files only stayed on the disk for several weeks. Hence the number of files which existed on **/backup** before the trace and were deleted during the trace is almost equal to the number of files which were created during the trace and still existed at the end of the trace. This indicates that the utilization of **/backup** stayed constant over a long period. A similar pattern is observed on **/users** which was kept below a certain level of utilization by users' self-discipline and by the system administrator's active intervention. On **Cello /**, in addition to a collection of binaries, libraries and other system files which were mostly read-only and long term files, there were a significant number of short and medium term files. It was found out that the file system was used to store temporarily news packets received from the network and thus a lot of short to medium term files appeared in our trace.

File system	Count
/	19598
/usr/local/src	16818
/tmp	9897
/mount/logs	1553
/users	38032
/backup	761
/usr/spool/news	406232
Total	492891

Table 3: Number of files on **Cello** that appeared in the trace

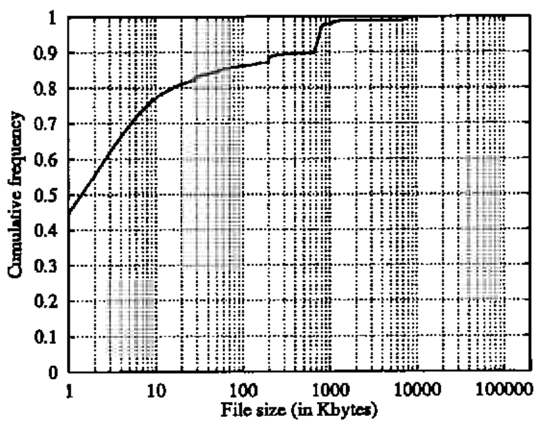

Figure 1: **Cello /** file size distribution

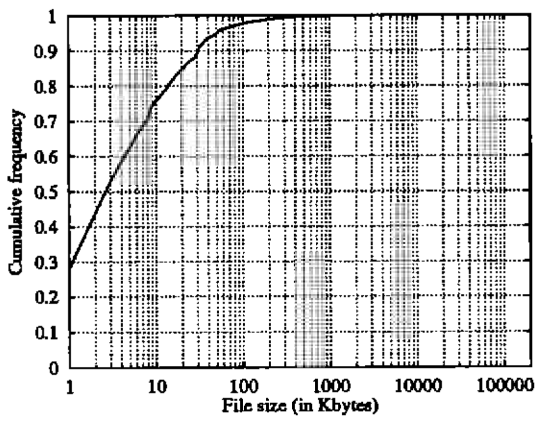

Figure 2: **Cello /users** file size distribution

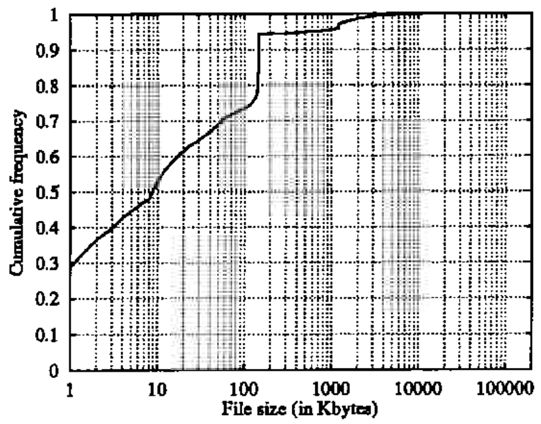

Figure 3: **Cello /tmp** file size distribution

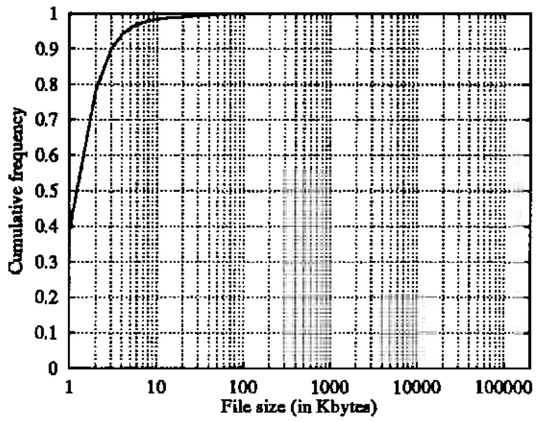

Figure 4: **Cello /usr/spool/news** file size distribution

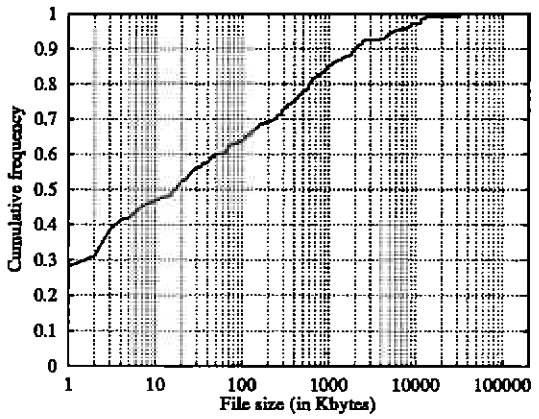

Figure 5: **Cello /backup** file size distribution

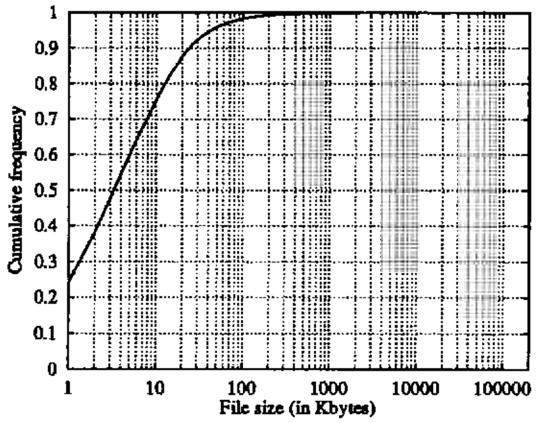

Figure 6: **Cello /usr/local/src** file size distribution

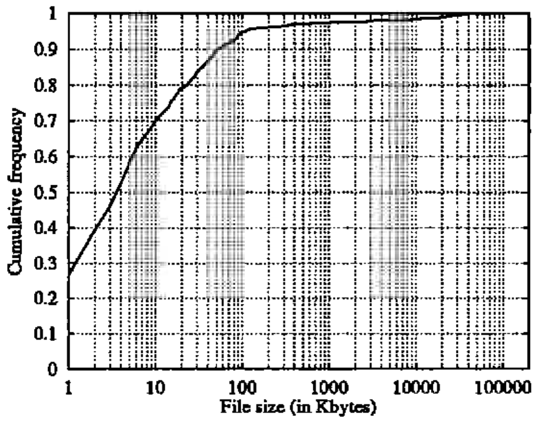

Figure 7: **Cello /mount/logs** file size distribution

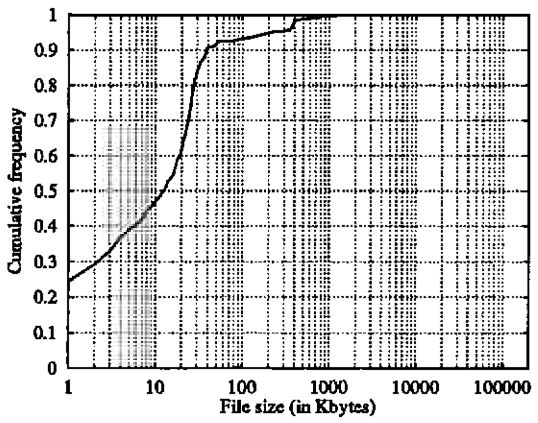

Figure 8: **Triangle /** file size distribution

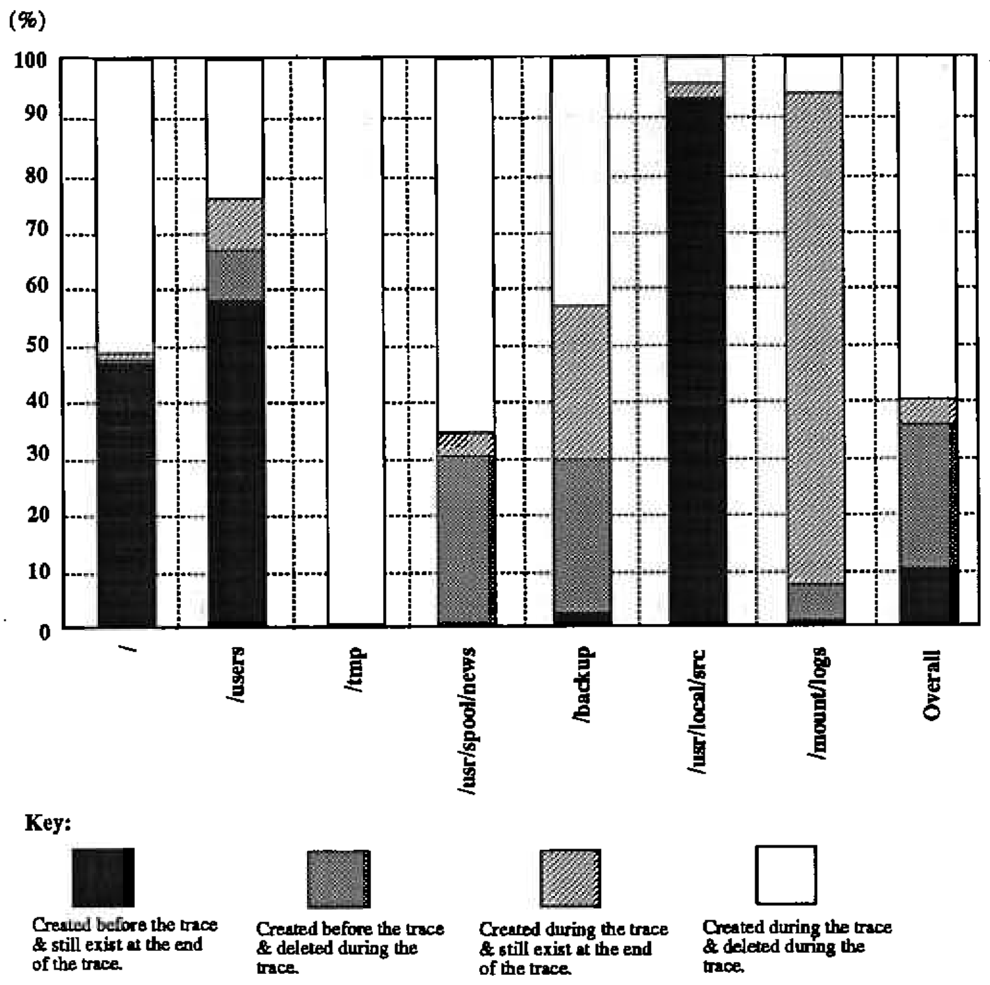

Figure 9: File system composition of **Cello** (in terms of number of files)

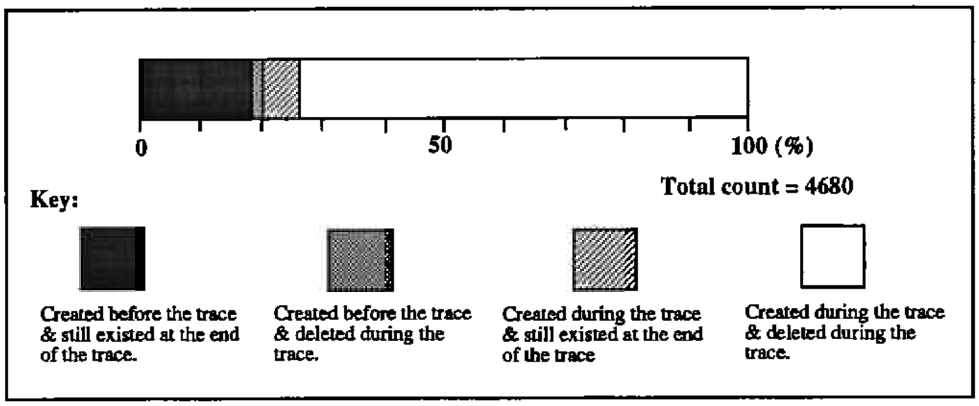

Figure 10: File system composition of **Triangle** (in terms of number of files)

5.4 File system activities

Figure 5.4 shows the file open counts on the seven file systems of **Cello**. Figure 5.4 shows the file open counts on **Triangle /**.

In each diagram, there are three stacked bars. The first bar (*Writable open count*) represents the number of times files were opened either write only or read and write. The second bar (*Read-only open count*)represents the number of times files were opened read-only. The third bar (*Read-only open count(with cache miss)*)is a subset of the second: when a file is opened read-only, IO may or may not be needed to fetch the data from the disk because it is possible that all the data required are already in the buffer cache. We are interested in how many times a file was opened read-only and has at least one disk IO when the data required was not in the buffer cache. This number is represented by the third bar. Each bar is subdivided into four regions and each region represents file opens for files in one of the categories defined at the beginning of this section. The bars in each graph are drawn in proportion for comparison and the actual value of each bar is shown on top of it.

It should be noted that *Writable open count* includes both file open write-only and file open read/write. This explains why in some cases, such as Cello /, *Writable open count* is higher than *Read-only open count(with cache miss)*.

As shown in the diagram, **/usr/local/src** is a read-mostly file system with a very small number of opens for write or read/write. On **/backup**, only long term files are opened read-only and only short or medium term files are open writable.

On **Cello /**, the long term files account for the majority of file opens read-only. However, in terms of *Read-only open count(with cache miss)*, i.e. those file opens with reads hitting the disk, the long term files on **Cello /** only account for a minor portion of the total open counts. In fact, the medium and short term files on **Cello /** actually dominate the disk traffic! However, it should be borne in mind that binaries on Cello / are fetched from disk with demand-paging and do not go through the normal file system interface. Hence this kind of reads from long term files is not accounted for in our study.

5.5 Files lifetime distribution

Figure 13 to figure 20 show the file lifetime distributions of the file systems studied. The file lifetime is defined as the time between the file was created to the time it was deleted with an unlink system call. Each figure is a cumulative frequency plot of the lifetime. In these graphs, we only include files that were created and were then deleted during the trace period. The number of files in each file system which belong to this category is shown in table 4.

File system	Count
/	10207
/usr/local/src	778
/tmp	9858
/mount/logs	103
/users	9299
/backup	332
/usr/spool/news	267849
Total	298426

Table 4: Number of files on **Cello** that were created and then deleted during the trace

Some features of note are:

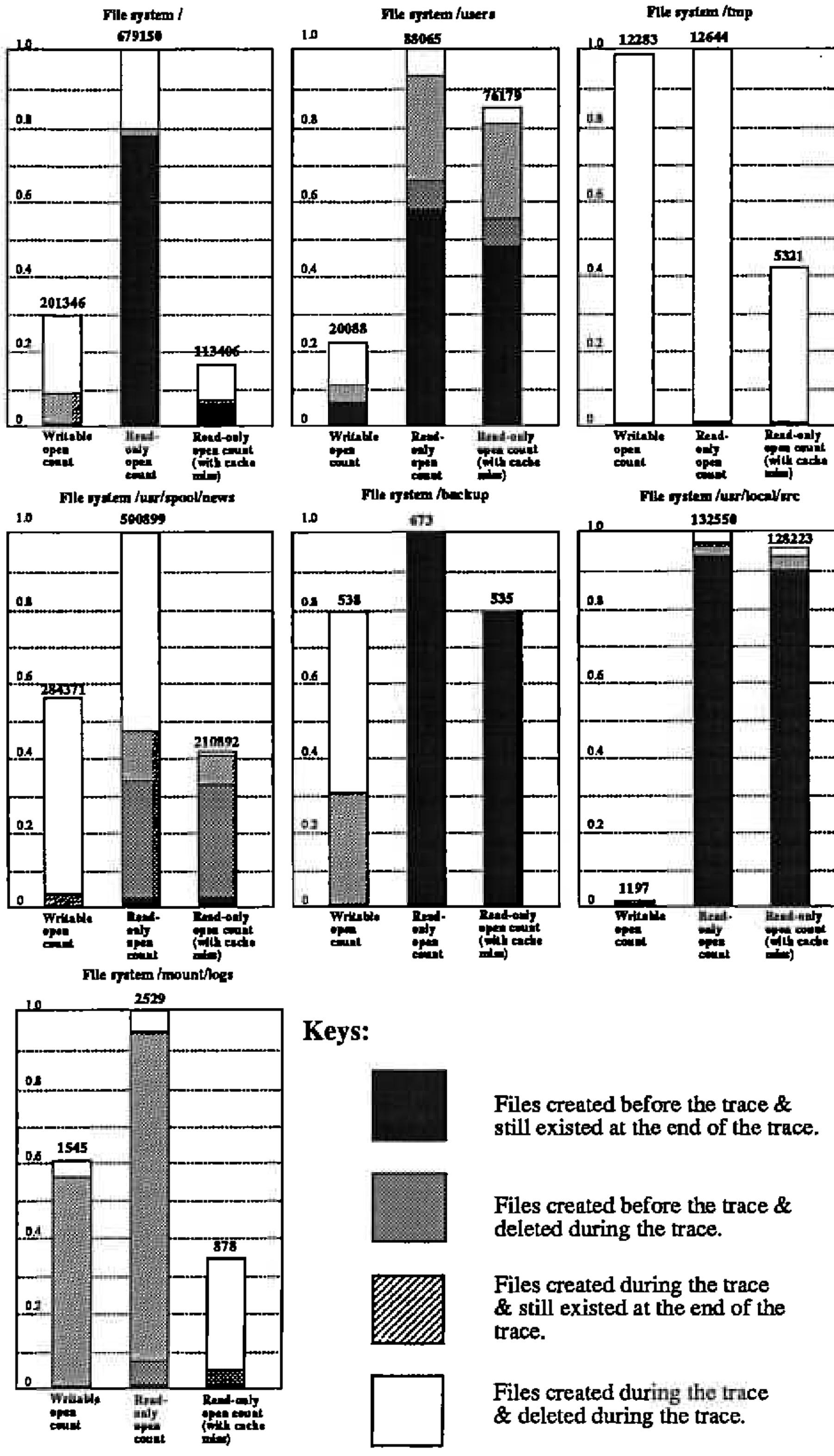

Figure 11: File system activities on **Cello**

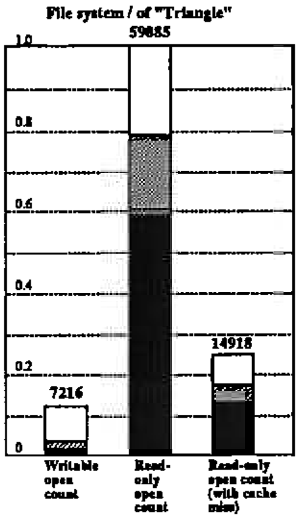

Figure 12: File system activities on **Triangle**

- In many of the graphs, quite a significant proportion of files have a life time of less than 1 second. This could be an artefact of the way the trace filter handles unlink records. Since in Unix, it is possible to create a file and then unlink (delete) it immediately but, as long as the file remains opened, a process can still read and write to it. Many Unix programs use this technique to create temporary files – these files would seem to have a very short lifetime but in fact they are not removed until they are closed.
- For **Cello /usr/spool/news**, the curve is nearly flat between 5 seconds and about 2 weeks (around 1.2 million seconds). There are two distinct usage of the file system: those files with short lifetime were likely to be working files of the news system whereas those files which existed for two weeks were the news items. The news system purged news items after two weeks.
- On **Cello /usr/local/src**, almost all the files that were created and then deleted during the trace had the same lifetime. This is probably because some large program packages (778 files) were installed and then probably found to be unsuitable and hence were deleted.

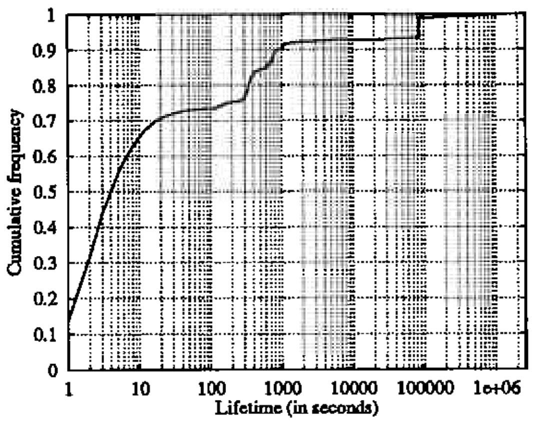

Figure 13: **Cello /** file lifetime distribution

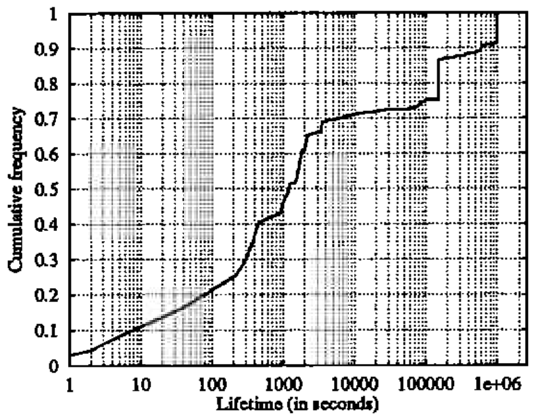

Figure 14: **Cello /users** file lifetime distribution

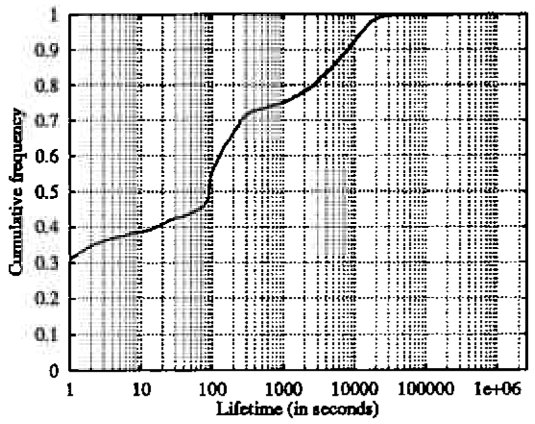

Figure 15: **Cello /tmp** file lifetime distribution

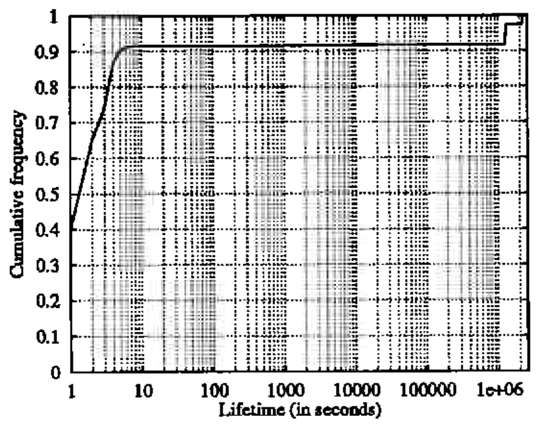

Figure 16: **Cello /usr/spool/news** file lifetime distribution

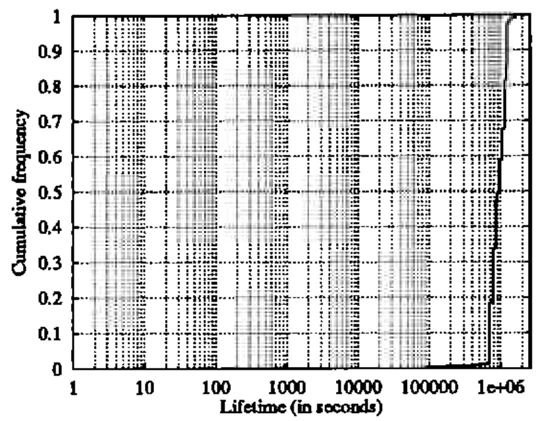

Figure 17: **Cello /backup** file lifetime distribution

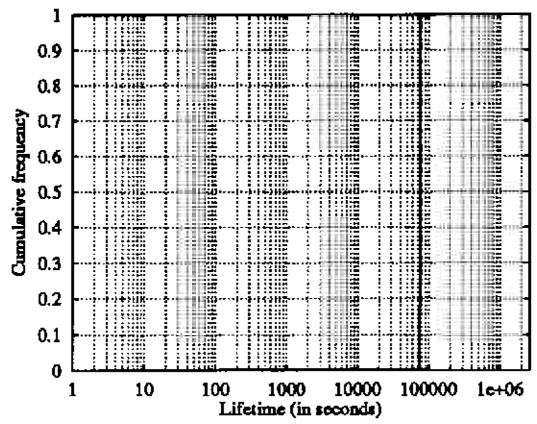

Figure 18: **Cello /usr/local/src** file lifetime distribution

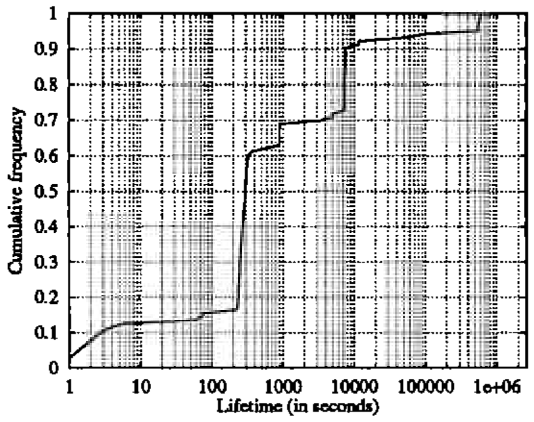

Figure 19: **Cello /mount/logs** file lifetime distribution

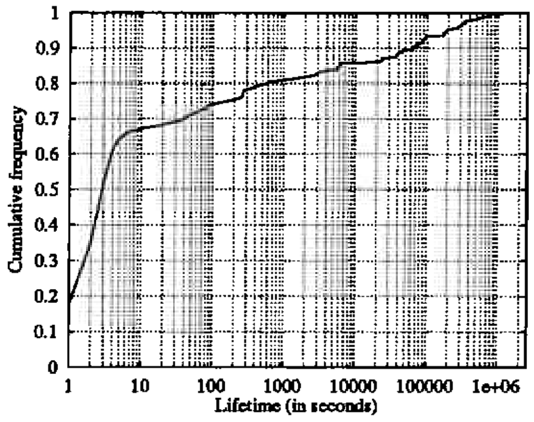

Figure 20: **Triangle /** file lifetime distribution

6 Simulation results

6.1 Introduction

In this section, the simulation results are presented. We chose three file systems to study and they were:

1. File system / of machine **Triangle**
2. File system / of machine **Cello**
3. File system **/users** of machine **Cello**

These file systems were chosen because these partitions contain a mixture of files with different lifetime, modification characteristics and different sizes.

Although we tried to simulate the real systems as closely as possible, we simplified our experiments by ignoring disk traffic due to access to superblocks, inodes and directories and by using simplier disk scheduling algorithm. Consequently we could not compare our simulation results with the real performance of the systems directly. Instead we ran a simulation for each file system with no replication and the results were used as the yardstick for comparison.

The columns of the tables used to display the results (eg. table 5 are as follows:-

No. of copies This is the number of copies kept for each file when it is replicated.

Seek distance This is the mean seek distance for processing each read and write request.

Disk queue length This is the average queue length for all disk IOs.

Disk read queue length This is the average queue length as seen by an incoming read request (i.e. not counting the new one).

Disk write queue length This is the average queue length as seen by an incoming write request (i.e. not counting the new one).

Read turnaround time This is the interval between the moment a disk read request enters a disk queue to the time the data is returned.

Write turnaround time This is the interval between the moment a disk write request enters a disk queue to the time the disk write finishes.

Overall turnaround time This is the weighed average of both read and write turnaround times.

When there is more than one copy, the value of *seek distance* is the weighed average of the mean seek distance of all disks. Similarly, the values of *disk queue length, disk read queue length* and *disk write queue length* are averaged over the corresponding values of each disk.

In all cases the results were checked with T-test. The changes in the mean values were found to be statistically significant with a 95% confidence level.

6.2 Read one out of N copies and write to N copies (RNWN)

In this series of simulations, a file is always replicated. A read request is dispatched to the replica with the shortest turnaround time. All updates are propagated to all copies (mirror disks). A write request returns as soon as at least one copy has been successfully updated. In the meantime, updates to other copies continue in the background, adding to the load on the disks.

Table 5 shows the results for **Triangle /**. The results for **Cello /** and **Cello /users** are shown in table 6 and table 7 respectively.

In all these cases, the mean read turnaround time drops as the number of copies increases. For **Triangle /**, the mean read turnaround time reduces by 8% with two copies and by 10.5% with three copies. With two copies, the value reduces by 32% for **Cello /** and by 26% for **Cello /users**. The value then levels off as the number of copies increases.

In all three cases, the mean write turnaround time also drops as the number of copies increases. With 3 copies, the value drops by 15% for **Triangle /**, 44% for **Cello /** and 22% for **Cello /users**. The value also levels off as the number of copies increases beyond 3.

The drop in turnaround times can be explained by the corresponding reductions in the mean *seek distance* and the *disk read/write queue length*. It should be noted that the mean disk queue length, which includes both reads and writes, increases with the number of copies. However, the **disk read queue length** and the **disk write queue length** follow a decreasing trend. This apparent discrepancy is due to the fact that the number of disk writes increases with the number of copies because all copies have to be kept up-to-date. As the number of disk reads remain constant, the ratio of writes to reads increases with the number of copies. Therefore the mean disk queue length increases as the disk write queue length is longer than the disk read queue length.

Triangle / and **Cello /users** are similar in that the seek distance, the disk read queue length and the disk write queue length decrease with the increase in the number of copies. The fact that this policy works better for **Cello /users** than **Triangle /** is because the disk read queue length in the non-replicated case is higher for **Cello /users** than **Triangle /**. As one of the effects of replication is the automatic sharing of load among the disks, the improvement in read performance is expected to be more significant with a longer disk queue and this is confirmed by our results.

The load balancing effect of replication is more evidence in case of **Cello /**. Unlike the previous two cases, the seek distance increases with the number of copies. However the disk read queue length drops from 0.45 to 0.09 and the disk write queue length drops from 3.96 to 2.85 when the number of copies increases from 1 to 2. Clearly the effect of load balancing in reducing the waiting time in a disk queue is more than offset by the increase in seek time.

6.3 Read one out of N copies and cut back to 1 copy on write (RNW1)

In this series of simulations, we examined the effect of dynamically changing the number of copies. A file, once created, is assumed to be replicated in the background and N copies would appear 60 seconds after its creation. However the number of copies is cut back to one, i.e. with only the master copy remaining, before an update gets performed. When no more update is performed, new copies are made in the background and N copies would appear 60 seconds after the file is last modified. We did not simulate the background replication and assumed that the disks have ample spare bandwidth to handle this background load. A read request is always dispatched to the replica with the shortest turnaround time. A write request only goes to the master copy.

The results of the simulations for **Triangle /**, **Cello /** and **Cello /users** are shown in table 8, table 9, table 10 respectively.

In all three cases, the mean read turnaround time drops as the number of copies increases. For **Triangle /**, the mean read turnaround time reduces by 14% with two copies and by 18% with three copies. For **Cello /users** the value reduces by 24% with two copies and by 25% with three copies. As for **Cello /**, the value reduces by 10% for two and three copies.

The mean write turnaround time also drops as the number of copies increases. With three copies, the value drops by 4% for **Triangle /**, 10% for **Cello /** and 3% for **Cello /users**.

It is useful to compare this set of results with that of RNWN. The difference between these two sets of simulations is that with RNWN the number of copies stays the same whereas in the present case it varies with the usage pattern. The present policy only makes new copies when the last time a file is modified is longer than 60 seconds ago. This would eliminate the replication of short life files and files which are modified fairly frequently. Also if files are used in such a way that it is first modified and is then read back shortly afterwards, the number of copies would always be one when it is read because the recent update has caused the system to cut copies. Hence the performance of the system for this type of usage would be the same as if no replication is done.

The improvement in read turnaround time for **Triangle /** is clearly more significant in the present case than RNWN. This policy scores better in both seek distance and disk queue length.

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	82.38	1.75	0.14	2.85	26.36	23.55	74.62	107.38	55.08	87.45
2	64.08	2.10	0.09	2.78	24.26	20.81	66.53	102.08	49.41	82.51
3	57.07	2.27	0.08	2.77	23.58	20.08	63.53	99.94	47.36	80.57
5	52.55	2.44	0.06	2.76	23.25	19.56	60.61	97.92	45.48	78.73
7	50.55	2.52	0.07	2.76	23.23	19.74	59.14	97.33	44.61	78.15

Table 5: **Triangle** / *Replication Policy:* Read 1 out of N Write N

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	16.91	0.74	0.45	3.96	28.22	29.61	150.76	102.35	38.38	53.07
2	24.67	0.52	0.09	2.85	23.92	20.69	102.05	101.02	30.40	41.28
3	25.71	0.59	0.08	2.47	23.31	18.36	84.71	102.85	28.32	38.40
5	27.75	0.71	0.06	2.17	22.68	16.84	73.88	104.19	26.93	36.88
7	33.85	0.99	0.07	2.02	22.51	16.34	69.87	103.96	26.44	36.23

Table 6: **Cello** / *Replication Policy:* Read 1 out of N Write N

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	79.73	1.82	0.38	5.80	27.42	31.42	126.22	299.62	53.54	162.36
2	55.56	2.31	0.11	5.36	20.35	19.40	105.29	292.92	42.81	156.09
3	51.36	2.76	0.09	5.24	19.54	17.84	98.89	291.16	40.52	154.51
5	47.18	3.33	0.07	5.16	19.04	16.89	94.27	288.30	38.93	152.60
7	44.71	3.68	0.06	5.12	18.89	17.08	92.19	286.46	38.27	151.51

Table 7: **Cello /users** *Replication Policy:* Read 1 out of N Write N

This suggests that avoid replicating short term files and frequently modified files does make the system performs better, though the cost of replication has not been simulated in this case.

On the otherhand, this policy does not work as well as RNWN for **Cello /**. At two copies, the disk read queue length is 0.26 with this policy and is only 0.09 with RNWN. Although the seek distance does go down with the increase in the number of copies, the reduction in seek time is not enough to offset the difference in queuing time. We postulate that the apparent drop in performance is because some files are first updated and then read shortly afterwards. Hence, as have been discussed, our present policy would have no effect on such files. The result is a high load on the "master" disk, which is the only one containing recently-updated files. On closer examination of the use of **Cello /**, there is a news spool directory where news packets received from the network are stored. These files are around 700K to 1Mbytes in size and are written once and are read shortly after they are written.

6.4 Read one out of N copies and cut back to 1 copy on write and choose copy to retain at random (RNW1CR)

With RNW1 policy, the original copy is retained when the number of copies is reduced. Therefore, in addition to reads to non-replicated files, all writes have to go to the same disk as well. On the otherhand, if the original copy is not kept and instead the copy to be retained is selected at random, this could spread out the load onto all disks. In this series of simulations, the policy is almost the same as RNW1 but during cutback, the copy to be retained is chosen at random.

The results of the simulations for **Triangle /**, **Cello /** and **Cello /users** are shown in table 11, table 12, table 13 respectively.

When compared with RNW1, the read turnaround time improves slightly and the write turnaround time improves modestly. The improvement in write turnaround time when compared with RNW1 is largest for **Cello /**. For instance, with two copies the write turnaround time reduces from 134.84ms to 111.07ms. The disk write queue length reduces from 3.50 to 2.84.

6.5 Scatter files among N disks

In the analysis so far, we have compared our results with the case when there is only one copy on one disk and we have seen that replication over a number of disks does have an effect on load balancing. However, it is possible that by scattering the files over the same number of disks, the load can also be spread across all the disks. To see what is the effect of scattering files over a number of disks, we performed this series of simulations in which files are scattered randomly among 2,3,5 and 7 disks. File replication is not done.

Table 14 shows the results of the simulations for **Triangle /**.

From the result, it is clear that scatter files among N disks does have an effect on reducing the queue length. For instance, the disk queue length reduces from 1.75 to 1.48 when files are scattered over two disks. As expected there is no reduction in the average seek distance. With the reduction in the disk queue length, the read and write turnaround times improves.

The improvement in read performance is not as much as in the case of RNW1, for instance, with three copies and RNW1, the mean read turnaround time is 21.57 and it is 24.93 with random scattering over 3 disks. However the write performance is better with random scattering than with RNW1. With three copies, the mean write turnaround time is 64.36 with random scattering and is 71.46 with RNW1. Although RNW1CR performs better than RNW1 in writes, random scattering still scores better in write turnaround time.

6.6 Combine file scattering and RNW1CR

We have seen scattering files among N disks alone performs better in writes than RNW1CR but worse in reads. In this series of simulations, we combined the two policies to see if we can gain

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	82.38	1.75	0.14	2.85	26.36	23.55	74.62	107.38	55.08	87.45
2	56.29	1.67	0.03	2.78	22.63	18.94	71.98	106.41	52.00	86.43
3	36.87	1.66	0.03	2.76	21.57	18.57	71.46	105.78	51.27	86.02
5	33.06	1.66	0.03	2.76	20.82	17.96	70.98	104.69	50.68	85.20
7	31.06	1.66	0.03	2.77	20.71	18.05	70.80	104.89	50.52	85.35

Table 8: **Triangle** / *Replication Policy:* Read 1 out of N and cutback to 1 on Write (RNW1)

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	16.91	0.74	0.45	3.96	28.22	29.61	150.76	102.35	38.38	53.07
2	13.15	0.53	0.26	3.50	25.39	24.93	134.84	110.98	34.47	50.03
3	12.39	0.53	0.26	3.49	25.33	24.93	135.28	111.19	34.45	50.16
5	11.64	0.53	0.26	3.49	25.25	24.81	134.78	111.32	34.34	50.05
7	11.15	0.52	0.26	3.48	25.19	24.74	134.56	111.32	34.27	50.00

Table 9: **Cello** / *Replication Policy:* Read 1 out of N and cutback to 1 on Write (RNW1)

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	79.73	1.82	0.38	5.80	27.42	31.42	126.22	299.62	53.54	162.36
2	38.79	1.61	0.15	5.69	20.89	24.41	121.84	298.49	47.58	161.18
3	40.99	1.60	0.13	5.68	20.61	23.79	121.41	296.81	47.26	160.29
5	30.68	1.60	0.13	5.69	20.42	25.13	122.37	300.50	47.38	162.37
7	29.60	1.60	0.13	5.68	20.32	24.26	122.30	301.57	47.28	162.80

Table 10: **Cello /users** *Replication Policy:* Read 1 out of N and cutback to 1 on Write (RNW1)

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	82.38	1.75	0.14	2.85	26.36	23.55	74.62	107.38	55.08	87.45
2	57.26	1.57	0.03	2.62	22.58	19.02	68.48	102.31	49.90	82.97
3	45.40	1.51	0.03	2.52	21.56	18.64	66.41	100.07	48.26	81.15
5	37.00	1.48	0.02	2.48	20.96	18.35	64.80	98.94	47.05	80.16
7	33.24	1.46	0.02	2.44	20.59	17.96	63.75	96.76	46.28	78.43

Table 11: **Triangle** / *Replication Policy:* Read 1 out of N, cutback to 1 on Write and choose copy to retain randomly (RNW1CR)

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	16.91	0.74	0.45	3.96	28.22	29.61	150.76	102.35	38.38	53.07
2	12.79	0.47	0.26	2.84	25.19	23.26	111.07	94.11	32.32	42.33
3	11.66	0.45	0.25	2.60	24.97	22.72	102.73	91.33	31.42	40.32
5	10.46	0.43	0.25	2.43	24.80	22.46	96.26	89.77	30.73	38.98

Table 12: **Cello** / *Replication Policy:* Read 1 out of N, cutback to 1 on Write and choose copy to retain randomly (RNW1CR)

No. of copies	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	79.73	1.82	0.38	5.80	27.42	31.42	126.22	299.62	53.54	162.36
2	39.48	1.52	0.14	5.35	20.69	23.84	115.69	293.60	45.81	158.01
3	34.96	1.49	0.13	5.29	20.36	22.88	114.04	291.96	45.13	156.94
5	31.95	1.51	0.13	5.34	20.40	24.73	115.60	285.76	45.57	154.28

Table 13: **Cello /users** *Replication Policy:* Read 1 out of N, cutback to 1 on Write and choose copy to retain randomly (RNW1CR)

better performance in both reads and writes. The only difference between this policy and RNW1CR is that when a file is created, a disk is chosen at random to store it.

Table 15 shows the results of the simulations for **Triangle /**.

From the results, it is clear that combining the two policies has achieved a read performance as good as RNW1CR and a write performance as good as random scattering.

6.7 Summary

Figure 21 is a plot of the read turnaround time of **Triangle /** vs the number of copies for all the policies we experimented with. The write turnaround time of **Triangle /** is plotted against the number of copies in figure 22. A similar set of graphs for **Cello /users** and **Cello /** are shown in figure 23 to figure 26.

No. of disks	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	82.38	1.75	0.14	2.85	26.36	23.55	74.62	107.38	55.08	87.45
2	79.36	1.48	0.08	2.43	25.15	19.19	64.98	95.03	48.85	76.85
3	79.03	1.45	0.07	2.40	24.93	17.97	64.36	96.46	48.40	77.73
5	81.19	1.47	0.06	2.23	25.16	16.32	60.44	91.54	46.16	73.45
7	82.63	1.38	0.04	2.28	25.32	15.27	58.88	89.23	45.29	71.44

Table 14: **Triangle /**, scatter files among N disks

No. of disks	Seek distance	Disk queue length	Disk read queue length	Disk write queue length	Read turnaround time (msec)		Write turnaround time (msec)		Overall turnaround time (msec)	
					Mean	S.D.	Mean	S.D.	Mean	S.D.
1	82.38	1.75	0.14	2.85	26.36	23.55	74.62	107.38	55.08	87.45
2	57.18	1.47	0.02	2.45	22.34	16.94	65.18	96.44	47.84	78.06
3	46.87	1.38	0.01	2.30	21.35	15.02	61.79	92.72	45.42	74.84
5	40.01	1.34	0.02	2.23	20.88	16.69	60.27	91.78	44.33	74.16
7	37.40	1.32	0.01	2.32	20.45	13.63	59.91	94.14	43.94	75.67

Table 15: **Triangle /**, Combine scatter files among N disks and RNW1CR

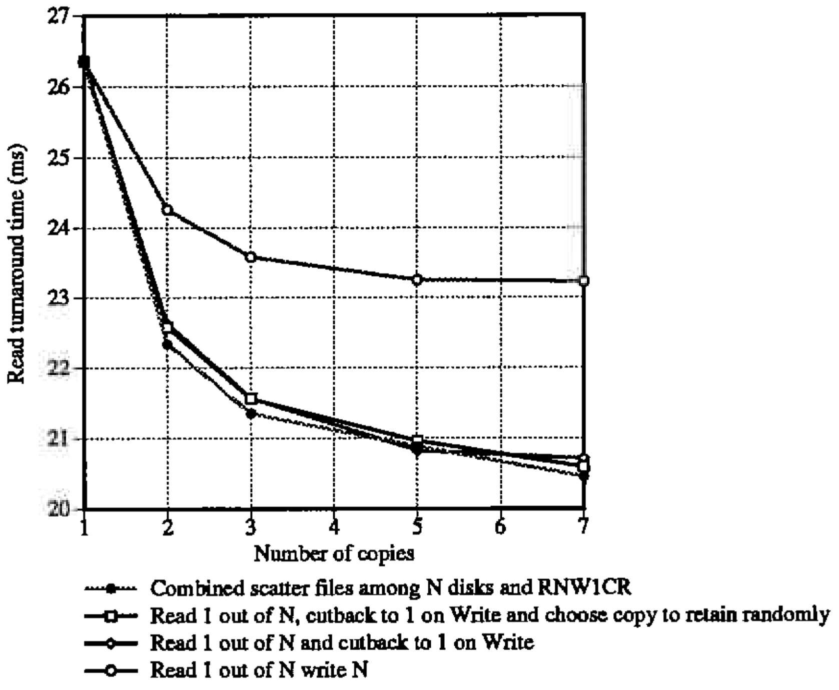

Figure 21: **Triangle** / read turnaround time vs number of copies

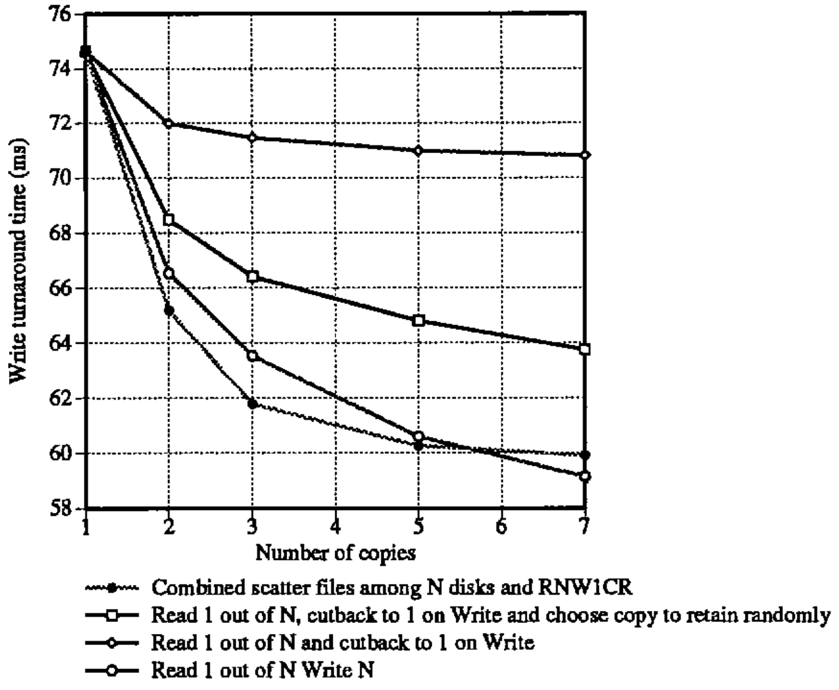

Figure 22: **Triangle** / write turnaround time vs number of copies

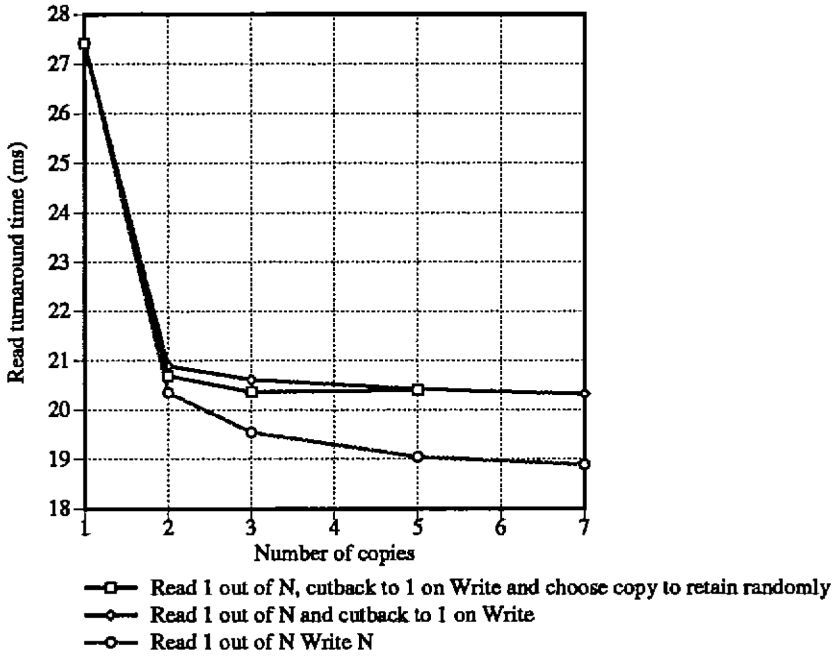

Figure 23: Cello / read turnaround time vs number of copies

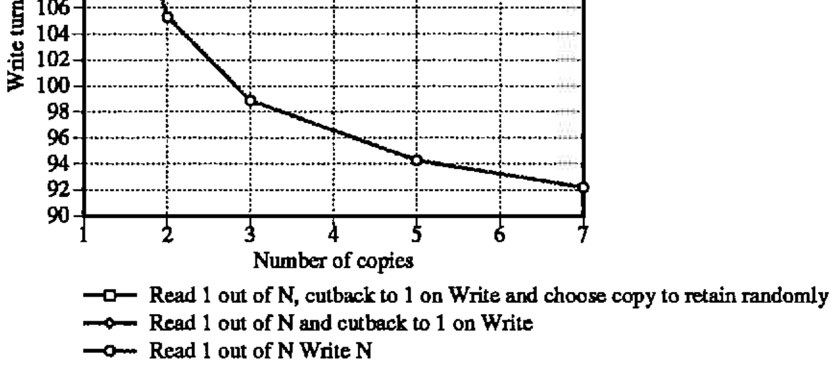

Figure 24: Cello / write turnaround time vs number of copies

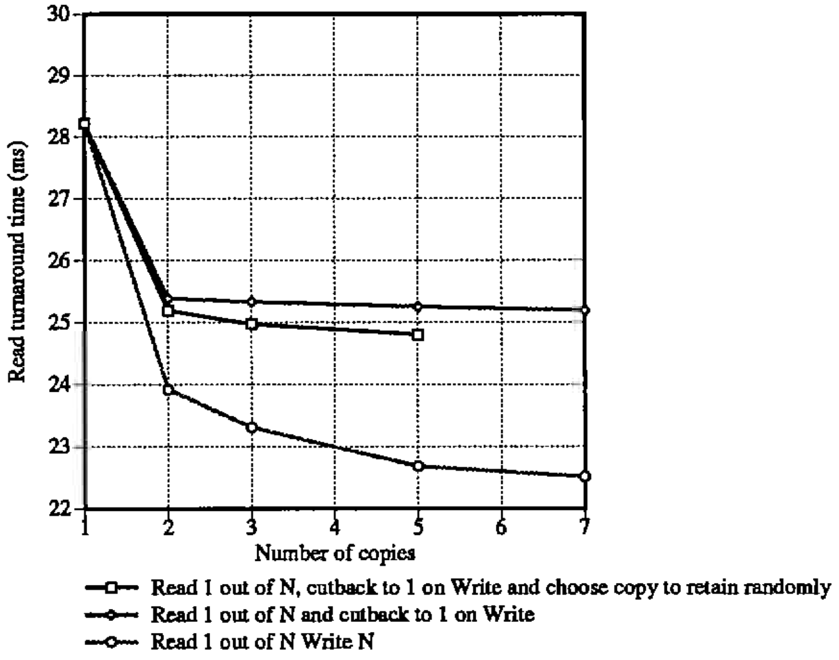

Figure 25: **Cello** / read turnaround time vs number of copies

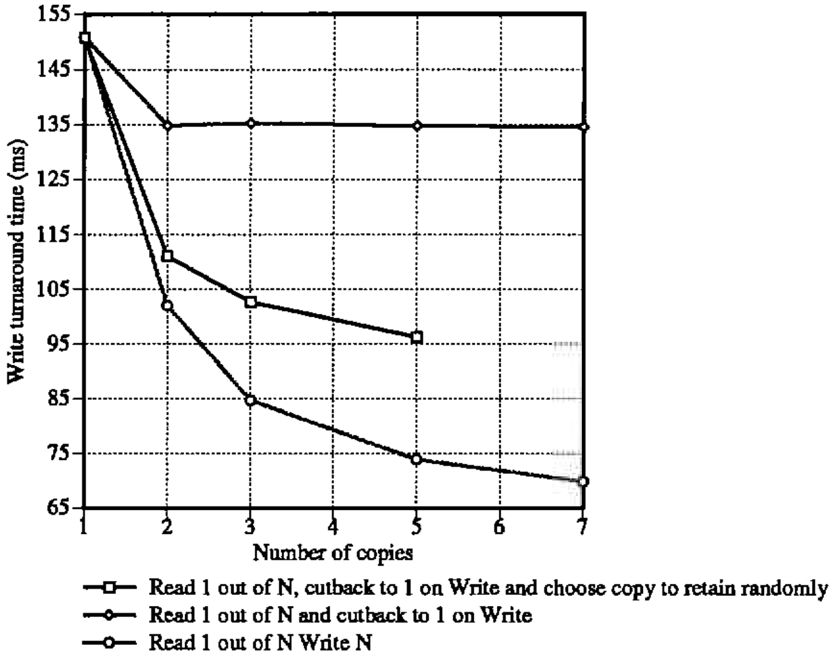

Figure 26: **Cello** / write turnaround time vs number of copies

7 Conclusion

Our study started with the hypothesis that replication can improve read performance through reduction in seek time, load balancing across multiple disks and parallel transfers. From our trace-driven simulations, we have shown that read performance did improve with more copies.

We compared "mirror disks" (RNWN) with dynamic replication (RNW1). Except in one case, dynamic replication worked better than "mirror disks". Moreover the exception can be explained by the special usage pattern of that particular file system.

We explored a number of alternatives in dynamic replication. While read performance remains almost the same in all cases, the write performance improves because write loads are spread among the disks available. It seems that introducing certain randomness in choosing which disk to place a copy when a file is first created and when the number of copies is cutback does help in spreading out the write loads and hence improving the write performance.

We noted that dynamic replication did reduce the disk queue length by sharing the load across multiple disks. It should be noted that the systems we studied were only lightly loaded. It will be interesting to see the effect of dynamic replication on heavier loaded machines since we expect the disk queue length on these systems would be longer and hence the opportunity for load-balancing would be larger.

We observed that dynamic replication did reduce the seek distance, however, it should be noted that better disk layout strategies such as rearranging data on disk according to the usage pattern could also bring about significant reduction in seek time. However, simply rearranging data on disk does not help very much in reducing the queuing delay and this is the area that our study have shown dynamic replication has a big potential.

In our study, only two or three copies were needed to bring about significant performance improvement. Further increase in the number of copies did not improve the performance much further.

From our simulations, we could get, through dynamic replications, 24% improvement in read performance with two copies (see **Cello /users** RNW1). One could question whether committing 100% more storage space to gain 24% performance improvement is a wise decision. However, it should be borne in mind that the storage space allocated is otherwise unused and we can always cutback the number of copies when the free disk space is tight. The real resources being used up is the CPU time and the bandwidth of the interconnects among the storage nodes, both of which are plentiful in a DataMesh.

References

[Cha90] Chia Chao. Hp-ux measurement system's build-in instrumentation points. Technical Report HPL-DSD-90-27, Hewlett Packard Laboratories, April 1990.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–162, 1979.

[MHS89] Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Technical Report 46, Systems Research Center, Digital Equipment Corporation, June 1989.

[MSC+86] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications ACM*, 29(3):184–201, March 1986.

[Pur87] T. Purdin. *Enhancing File Availability in Distributed Systems (The Saguaro File System.* PhD thesis, University of Arizona, 1987.

[SKK89] M. Satyanarayanan, James J. Kistler, and Puneet Kumar. Coda: A highly available file system for a distributed workstation environment. Technical Report CMU-CS-89-165, School of Computer Science, Carnegie Mellon University, July 1989.

[Sov84] Liba Sovobodova. File servers for network-based distributed systems. *Computing Surveys*, 16(4):353–398, December 1984.

[Tre88] G. Winfield Treese. Berkeley unix on 1000 workstations : Athena changes to 4.3 bsd. In *USENIX Winter Conference Proceedings*, 1988.

[Wil89] John Wilkes. Datamesh-scope and objectives: a commentary. Technical Report HPL-DSD-89-44, Hewlett Packard Laboratories, July 1989.

[WPE+83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The locus distributed operating system. *ACM Operating System Review*, 17(5):49–70, October 1983.