Combining the Concepts of Compression and Caching for a Two-Level Filesystem

Vincent Cate and Thomas Gross
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Storage systems have always attempted to use properties of the files that are stored in the system to optimize access time, capacity, and/or cost. Compression exploits patterns within files, and file migration and file caching exploit file access patterns, but the combination of these concepts has not been reported on before. We discuss here the effectiveness of a filesystem that integrates caching and compression to provide two levels of file storage on disks. This investigation is based on measurements that were collected on nine computers at three different sites. The data indicate that automatic compression of least recently used files doubles the amount of data that can be stored on a given disk system, while incurring only a slight performance cost.

1 Introduction

The design of a storage system for a computer attempts to balance cost, access time, and storage capacity so that an effective filesystem can be provided for the user. To this end, the storage systems of all computers contain a storage hierarchy, which combines storage types with different cost and performance attributes. By combining two types of storage, a high-capacity, slow, and inexpensive store and a small, fast, and expensive store, we can increase the total storage capacity for a given budget, compared to a system that uses only the faster but expensive storage. However, if we are able to exploit the locality of access to files, then we obtain a storage system with an average access time that is close to the access time of the fast and expensive store, whereas the storage capacity and average cost is determined by the capacity and cost of the slow and inexpensive storage.

Disks play a central role in the storage system of today's computers, and lately, several research projects have turned their attention to the design and evaluation of disk systems[13, 15]. Most of this research is motivated by the needs of high-end systems or supercomputers, where applications often demand high I/O bandwidth as well as low access latency. However, latency and/or bandwidth are not always the most important factors. In some environments, the objective is to store as many bytes as possible on a disk to keep the storage cost down, and this has been recognized by other researchers as well (e.g. cost is the motivation for picking a RAID-3 design over a RAID-1 design[13]).

File compression increases the amount of user data that can be stored on a disk but incurs an overhead for compression and decompression. Several algorithms have been developed in recent years that hold great promise in terms of their speed and how much they compress data [2, 21, 9, 3], but so far, methods for incorporating automatic compression into hardware or the operating system have not been widely used.

File migration is one way to exploit long-term locality in access patterns to files. Files that are accessed frequently stay in the faster store (usually disks), whereas infrequently accessed files are removed from the faster store and moved onto slower store (usually tapes, either handled in an automatic tape storage system or handled by operators). Depending on the system, a file may be considered as "infrequently accessed" if it has not been accessed for a time span measured in days or weeks. The caching of whole files is based on a similar idea but usually exploits short-term locality: when accessed, files are copied from the slow store (e.g. a file server with limited bandwidth) into the fast store (e.g. a local disk). In this case, the access speed is improved since caching reduces the amount of data each client transfers from the server,

This research was supported in part by Hewlett Packard and in part by the Defense Advanced Research Projects Agency (DOD), monitored by DARPA/CMO under Contract MDA972-90-C-0035.

thereby reducing the load each client places on the server [6].

There are two differences between file migration and file caching. The first difference is the frequency of file movement; with file caching, a file can be moved multiple times a day, whereas file migration is more long-term oriented. The other difference is the place where each file has its "home" location: in a system with file migration, all files are (conceptually) stored in the fast store and only moved to the slow store when certain conditions are met. In a system with file caching, files reside in the slow store and are moved to the fast store when necessary. But common to both is that they increase the storage capacity (compared to a system that only includes the faster storage), and that they incur some overhead for accessing as well as managing the files. Furthermore both are implemented in software, which has an impact on performance but also provides the opportunity to modify, measure and evaluate these methods.

Since file compression and file caching/migration both aim to increase the storage capacity of a system, we want to investigate what benefits can be obtained by combining the two: The disk level of the memory hierarchy is split into two levels. The faster level holds uncompressed data, and the slower level holds compressed data. The compressed data have decreased storage costs (less disk space is required) but have slower access times. In such a two-level file system, files move between these faster and slower level in much the same way that files move between disk and tape or between a local and a remote disk in other storage hierarchies. The least recently used files are considered "inactive" and compressed, the more recently used files are called "active" and kept uncompressed. Every night some of the active files are reclassified as inactive and compressed.

We are especially interested in how such a technique can be used in our day to day computing environment, which consists mainly of networked Unix[1] workstations and servers. Our environment does not include an automated tape library, and we do not have the resources to hire a large staff of operators. There are two issues we are concerned about: First, neither file migration nor compression has caught on in isolation in our environment, so we want to insure that the combined method is suitable for practical deployment. Second, we want to investigate the overall effectiveness of such a combined method. That is, we want to evaluate

- How much additional storage can be obtained, compared to a base-line system without compression and caching?
- What is the cost (in added access time) of the method?

[1]Unix is a trademark of AT&T Bell Laboratories

Input File	Size in KBytes	Compress CF	Frstword CF	PPMC CF
comp.c	40	2.07	2.94	3.21
vmunix	1,494	1.95	1.85	2.82
bitmap	1,056	5.76	6.27	5.73
paper.ps	98	2.25	3.04	3.55
paper.tex	45	2.13	3.15	3.11
netnews	318	2.30	3.23	3.35

Table 1: Compression factors

To answer these questions, we instrumented nine computer systems and collected data over time spans ranging from a few weeks to several quarters.

2 Background

Before we discuss our measurement technique, we briefly review in this section prior work on file compression as well as file caching and migration.

2.1 Compression algorithms

The idea of data compression has been around for a long time [7]. Recently there has been renewed interest in this field.

Welch [21] describes a fast implementation of the Lempel Ziv algorithm [23]. Welch's paper lead to the widely used Unix *compress* program. This program is fast and does a good job of compressing a wide range of files.

Bell [2] compares a number of different algorithms on a wide range of data types. Of the 14 types of input data, only 3 file types cannot not be compressed by at least a factor of 3 using the PPMC algorithm by Moffat [10]. Of the three files that cannot be compressed by at least a factor of 3, one contains floating point data, and the two others are executable binaries. These binaries are CISC binaries, which do not compress as much as RISC binaries, yet compression factors of 2.97 and 2.13 are obtained. Floating point data, on the other hand, is notoriously difficult to compress.

Table 1 shows compression factors for three compression programs: Unix *compress*, *frstword*, and *ppmC*. The last two have been developed by Moffat and correspond closely to the algorithms presented in [9] and [10] respectively. The data storage parameter on the *ppmC* program was set to its maximum value of 896 KBytes. The names of the input files reflect their content. The "comp.c" file is the C source code for the Unix "compress" program. The "vmunix" is a Mach kernel for a Sun-4. The "bitmap" is a 300 dpi image of a page of text. The "paper.tex" and "paper.ps" files are Latex and Postscript versions of one revision of this paper. The “netnews” file was made by combining many netnews posts.

Table 2 compares the speed of these same three compression algorithms on different computers; it lists the compression and decompression rates for the “netnews” input data set in Table 1. The decompression speed (in KBytes/sec) is reported using the size of the uncompressed file. We found that the decompression speed measured on this input set was representative of the decompression speed for each program on a mixed workload.

2.2 System integration of file compression

File compression has the potential to significantly increase the storage capacity of a disk. A number of approaches have been used to try to realize this potential. Each of these approaches has significant drawbacks.

2.2.1 Special hardware

One approach is to add special hardware to compress all data before they are stored to disk without degrading performance [22]. Two disadvantages to this approach are that special hardware increases system cost, and that special hardware is less flexible than software compression. Software compression makes it possible to select among many special compression algorithms so as to achieve the highest possible compression on each file and allows for an evolution of compression algorithms.

Ideally we could just add some hardware to compress data going to the disk and decompress data coming from the disk. However, this has not worked. The problem with this approach is that the operating system needs to allocate storage on a disk with a fixed number of fixed-sized sectors but it cannot know how much the data will compress. If disks did not need to store fixed-sized blocks of data, or if the disk were to perform storage allocation, this approach could be made to work. However, neither of these is likely to happen.

A number of systems compress files when storing them on tape. Special compression hardware has been added to the tape drive [1], resulting in significantly increased tape capacity. In a number of mass storage systems (for example [4]), files are compressed when they are moved from disk onto tape. This approach reduces the number of tapes needed for migration, archive, or backup. While useful at lower levels of the memory hierarchy, this approach does not improve the disk level.

2.2.2 Application specific

Another approach is not to modify the system in any way but to modify each application separately. For example, one group has modified the Unix online manual “man” so that it stores its data in a compressed format. This approach has been useful, but it also has drawbacks. The main problem with this approach is that it is necessary to modify each application. An additional problem is that it results in significant performance degradation: every time a file is accessed, it is necessary to decompress it, even when the file is used several times during the same day.

2.2.3 Operating system modifications

Another approach is to modify the operating system to compress *all* files, as suggested in [14]. With this approach, no modifications to applications are needed. The main disadvantage of this approach, however, is its poor performance. Without special hardware, it takes longer to read and decompress a file than to simply read it, and if this penalty must be paid for all files on all accesses, the cost becomes prohibitive.

An operating system could allow users to mark certain files and then always compress these files when they were closed and decompress them when they were opened. This approach results in significantly reduced performance for these files, since they must be compressed every time they are closed and decompressed every time they are opened. Furthermore, it requires the cooperation of the individual user.

2.2.4 Manual compression

Another approach, used in [14], is to provide utility programs and let the users compress and decompress their files manually. This approach has been quite successful and is in widespread use today. Programs make it as simple for users as typing *“compress file”*. However, since it takes some extra time and effort to compress and decompress files, users tend to compress only a small fraction of their total files. Another obstacle keeping users from compressing their files manually is that the applications programs that the users invoke cannot handle compressed files. For example, if the mail reading program cannot handle compressed files, users will not compress their mail files.

2.3 File migration and caching

File migration and file caching both exploit locality in the file access patterns. The differences between migration and caching are not really relevant for our investigation, and we will therefore consider these two together.

2.3.1 Migration

Many studies relevant to file migration have been reported in the literature [20, 17, 18, 16, 8]. Although the locality depends on the workload, the same trace was used in three of these papers [20, 17, 18]. This trace was obtained by instrumenting an editor at the Stanford Linear Accelerator

Machine	Compress		Frstword		PPMC	
	Compress	Decompress	Compress	Decompress	Compress	Decompress
VAX 11/780	14	27	3.3	3.8	1.9	1.7
VAX 11/785	22	43	5.0	5.7	3.1	2.8
Micro VAX 3	38	76	8.6	10	5.4	4.8
HP 9000/840	76	160	6.8	6.7	11	9.1
HP 9000/845	110	400	7.0	7.2	25	22

Table 2: Compression and decompression speed in KBytes/sec

Center (SLAC). The trace covers about a year, starting in August 1974. The users on this system had a small amount of disk space (70 KBytes to 1.4 MBytes), and they manually migrated their files to tape.

Over the last 15 years, the average computing environment has changed significantly. Users move files to tape primarily for archival storage, and the size of the data set kept on line has grown. Our hypothesis is that the file usage characteristics have changed since these earlier studies so that those data are no longer a suitable basis for design decisions. The data that we report in Section 4 support this hypothesis.

2.3.2 File caching

File caching has been used recently in distributed systems[6, 12]. In a distributed system, files are stored in some central facility whereas the computation takes place in a workstation on the user's desk. To overcome the access latency of the network that connects the workstation with the file server, as well as to reduce the overall load on the network, the local disk of the workstation holds the most recently accessed files.

While quite effective in meeting its design goal of masking the network latency[6], the file caching on the local disk attached to the user's workstation does nothing to improve the storage capacity of the central file servers.

3 Temporal locality of files

To evaluate the overall performance of a two-level filesystem that combines caching and compression, we need to know what percentage of all accesses is directed at the fast storage, and how often will it be necessary to retrieve a file from slow storage. We are therefore led to ask: what is the long term temporal locality of file access patterns?

3.1 Instrumentation

We collected data on both System V and 4.3 BSD versions of Unix. The monitoring program makes one pass over the inodes to determine how long ago each file was accessed. This tends to take about 5 minutes per GByte (billion bytes) of disk. Since this is done late at night, monitoring a system does not impact its performance significantly.

The program that collects the data opens the raw disk in read only mode. It is a simple program consisting of less than 300 lines of C code. Since this is the only program that needs any special privileges, it is relatively easy to convince a system administrator that collecting data is not going to damage the system. (Unlike other methods, this method does not require any kernel modifications.) Because of this it was easy to collect data from a number of systems. Therefore, unlike some studies that are based on trace data from only one system, we have collected data from nine systems.

With this technique, only a small (and therefore manageable) amount of data must be captured to record the system behavior. This is vastly superior to gathering traces, which can be quite large. The small summary information collected on remote machines can be sent via electronic mail to another machine, so a number of remote systems can be monitored with little effort. Mailing the data off during the night ensures that monitoring has a minimal impact on the system. Furthermore, the compactness of the data allows us to keep many months worth of data from many machines without requiring unreasonable amounts of storage.

3.2 Data collection method

The instrumentation described above provides us with an easy way to collect manageable amounts of data that indicate how well a least-recently-used (LRU) migration/caching algorithm works on a given system. Data can be collected for one month, and accurate miss rates can be measured for that month, even for a cache large enough to hold all files accessed in the last month. In fact, given data on only two adjacent days, we can tell how well LRU performs on that day for cache sizes corresponding to the amount of data accessed in the last 24 hours, 48 hours, and so on.

Using LRU as the caching strategy allows us to establish an upper bound on the amount of migration between the two levels of storage and hence a bound on how much compression/decompression must be done. If the caching is implemented in a system, this implementation can do at

Age (days)	# Files on Start Day 1	# Files on Start Day 2	# Accessed During Day 1
0 - 1	7195		6692
1 - 2	* 1000	503	* 300
2 - 3	1247	* 700	707
3 - 4	517	540	121
4 - 5	404	396	89
5 - 6	491	315	53
6 - 7	655	438	52
7 - 8	28	603	1
8 - 9	45	27	10
9 - 10	293	35	28

Table 3: Using histograms to calculate number of files accessed

Age (days)	KBytes on start of Day 1	KBytes on start of Day 2	KBytes accessed during Day 1
0 - 1	88954		86975
1 - 2	11284	1979	8747
2 - 3	24881	2537	10691
3 - 4	7343	14190	963
4 - 5	5582	6380	3028
5 - 6	6764	2554	597
6 - 7	19966	6167	615
7 - 8	2904	19351	2056
8 - 9	* 764	848	* 63
9 - 10	5457	* 701	331

Table 4: Using histograms to calculate total KBytes accessed

least as well as LRU since we can easily implement LRU. Modifications of the LRU strategy can produce even better results (see Section 4.2). Therefore, our measurements present an upper bound on the amount of data that must be compressed/decompressed in practice.

By taking snapshots of a filesystem exactly 24 hours apart and comparing them, it is possible to tell how many files of each age (in days) were accessed. In fact, it is only necessary to keep a few histograms counting parameters such as the number of files of each age and the total number of bytes for files of each age.

This data collection method requires an “age” or “time since last used” for each file. Unix keeps 3 times for each file - the time last read, the time last written, and the last time the inode was last modified. Since write operations can be buffered, the time of last read is used to determine the age of files.

Table 3 illustrates how the number of files accessed in each age group is calculated from the histogram data. For example, at the start of Day 1, there are 1,000 files that were last accessed between 24 and 48 hours ago. At the start of Day 2, there are 700 files that were last accessed between 48 and 72 hours ago. From this we can conclude that 300 of the original 1,000 files were accessed during the 24 hour period between the start of Day 1 and the start of Day 2.

Using totals for the number of bytes in files last accessed at each age on two days, it is possible to calculate the number of bytes accessed during the period between those two days. The right column in Table 4 was generated in this way. At the start of Day 1, there are 764 KBytes of data were last accessed between 8 and 9 days ago. At the start of Day 2, there are only 701 KBytes of data accessed between 9 and 10 days ago. This indicates that 63 KBytes of the original 764 KBytes of data that started Day 1 with this age were accessed during Day 1.

For a given cache size, we compute the age of the data still in the cache by adding up the entries in the left column, starting with age 0-1 days, until we reach an age N such that the data with age less than or equal to N days still fit in the cache but the data of age $N - (N + 1)$ do not fit any more. This establishes a "dateline" N for this cache size. The right column in Table 4 shows the amount of data accessed for each age. From this we calculate the amount of data that is not in a cache that holds the last N days of accessed data. This is computed by adding up all of the numbers below row N in the right column. For cache sizes that are larger than is needed to hold data for the last N days but smaller than for $N + 1$, interpolation is used. Thus we can determine, for each cache size, how much data would not have been in a least recently used cache on a given day and would have to have been decompressed.

3.3 Limitations of the data collected

One problem with this method of data collection is that it does not indicate how well caching works for caches smaller than the amount of data accessed in one day. With a cache that is too small to hold all of the files accessed on a single day, it is possible to miss multiple times on the same file during one day. Since there is no information about the order of accesses on a given day or the number of times a file was accessed, this situation can’t be analyzed. Hence, from our histogram data, we can only predict how well this approach works for larger cache sizes.

Another limitation of our data is that a file deletion is counted as a read. In the previous example, on Day 1 there are 1,000 files between 24 and 48 hours old and then on Day 2 there are only 700 files between 48 and 72 hours old. We know that 300 files have been touched in some way, but they may not have been read. So the data collected puts an upper bound on the number of files read.

Since data are only collected during the middle of the night, one might be concerned that during the day there was much more disk space in use. Three systems were monitored and this was found not to be the case. For 2 weeks the disk usage was recorded at 5 minute intervals and daytime total disk space in use at any one time was found to be less than 1% above nighttime usage each day for each machine[2]. It will be seen later that this level of precision is sufficient to justify the conclusions of this investigation.

A final limitation is that the clock on the machine being monitored needs to be more or less correct. If the clock is within an hour of being correct and data is collected at 4:00 a.m. there is little error due to the clock. If the clock jumps by a day or two files can be counted as read when they have not been or even look like negative reads. One of the systems monitored had such a problem and the data for the affected period was not used.

4 Results

To evaluate a caching strategy it is important to focus on the absolute amount data that must be decompressed and the amount of time that users are delayed while waiting for the decompression. The miss ratio is only relevant in that it provides a handle to compute these parameters of interest. To obtain the absolute delay introduced by the two-level file storage, we first calculate the total amount of data that needs to be decompressed for each percentage of the data kept uncompressed. Next we use the speed of the processor on decompression and the number of users to calculate an average delay per user per day.

Figure 1 shows how many KBytes the system B.GP.CS has to decompress per day as a function of the percentage of the data kept uncompressed. This figure is on a log-log scale. For example, if a quarter of the file data is kept uncompressed, on the average about 4.1 MBytes of data must be decompressed each day. If the system keeps half of the file data uncompressed, about 1.6 MBytes must be decompressed.

These data are then used to compute how much CPU time per user this decompression takes, as a function of the percentage of data kept uncompressed. The program that monitors a system keeps track of how many different user IDs are seen for the active files. This will count every user who reads at least one of his or her own files during the day (possibly the .login file). This method also counts references to files owned by system accounts, such as “mail” or “root”, as a user login; therefore, the number of system accounts was manually subtracted from the measured number of users. This gives a reasonably accurate approximation of the number of people that used a machine during a given day.

Figure 1 also shows how much CPU time is spent *per user* on decompression for the B.GP.CS system if the *compress* algorithm is used. The KBytes/day from the left is divided by the number of users and the decompression speed for B.GP.CS to get the CPU seconds/user per day. The real time delay depends on the system load. In a system with 50% CPU utilization, the real time would be about twice as long as the CPU time. The overall contribution to CPU utilization is negligible for computers with reasonable loads.

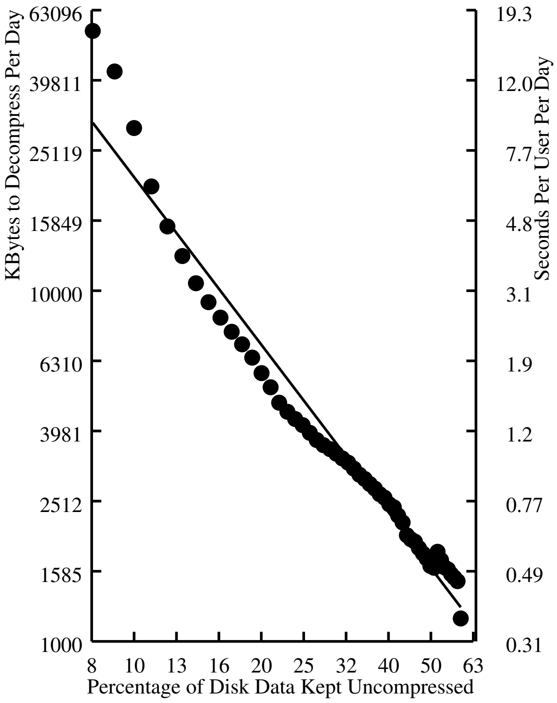

Figure 1: Cost vs. cache size for B.GP.CS

Figures for the other systems are attached in the appendix. As in Figure 1 the time per user is computed using the decompression speed for the *compress* algorithm on that machine as shown in Table 2. As can be seen, all graphs follow roughly the same pattern. The most peculiar system we monitored is HPCUPT1 (see appendix). This system is used almost exclusively for reading netnews. On this system, most users do not log in for a long time, so the average number of users logged in at once is much less than the average number of users in a day. The netnews files are heavily used but do not live to become old files, since they are soon deleted. This peculiar usage explains why the graph for HPCUPT1 seems to need two separate interpolation lines.

Table 5 describes the systems monitored. All of these systems run Unix. The three systems at Carnegie Mellon are used by computer science graduate students and faculty. The other systems are used in research and development organizations.

[2]This is not surprising since on many machines only around 10% of the data on disk are accessed on any given day.

Location	Machine	Processor	Max data in MBytes	Avg users in one day	Days monitored	Average % of data touched per day
AT&T Bell Labs.	CORONA	9000/845	3,800	71	57	6.6
Hewlett-Packard	HPCUPT1	9000/840	955	110	97	13
Hewlett-Packard	HPDBLPY	9000/840	1,920	22	66	12
Hewlett-Packard	HPPERF1	9000/840	2,210	61	68	3.6
Hewlett-Packard	HPISOI0	9000/840	1,527	32	13	13
Hewlett-Packard	HPISOI2	9000/840	552	8	18	5.4
Carnegie Mellon	SAM.CS	VAX 11/785	690	25	226	5.6
Carnegie Mellon	B.GP.CS	Micro-VAX 3	952	43	257	8.1
Carnegie Mellon	UNH.CS	VAX 11/780	521	8	267	5.0

Table 5: Summary of systems monitored

Machine	MBytes per day at 25% uncompressed	KBytes per user per day if 25%	Sec/user for compress	Sec/user for frstword	Sec/user for PPMC
CORONA	36	510	1.3	70	23
HPCUPT1	5.5	49	0.3	7.3	5.4
HPDBLPY	50	2,300	14	340	260
HPPERF1	17	280	1.7	42	31
HPISOI0	54	1,700	11	252	190
HPISOI2	1.8	220	1.4	33	24
SAM.CS	1.5	61	0.7	6.1	13
B.GP.CS	4.1	96	1.3	9.6	20
UNH.CS	1.0	130	4.9	35	78

Table 6: Summary of results for systems monitored

In Table 6 we combine the data presented in the graphs and tables to determine the average decompression delay the users would have experienced if the systems monitored had been compressing all but the most recently used 25% of the data. The second column of Table 6 is obtained from the data plotted in Figure 1 and the appendix. The third column is calculated by dividing the values in the second column by the average number of users on that system. Next we calculate the time to decompress this much data using each of the 3 algorithms from Table 1. As shown in the graphs, keeping less data uncompressed results in more decompression each day and conversely keeping more uncompressed results in less decompression.

As is evident from the data presented, how well the combination of caching and compression works varies from system to system. It should be noted that this approach will not work on a system that uses a small number of large files. In particular, a system that uses one large database file will not gain anything from this approach. The HPDBLPY system is in the database laboratory at Hewlett-Packard and was running database tests and large "makes" during the time it was monitored. Of the machines monitored it is the one least suited to for such a two-level file system.

In the systems monitored, the *compress* program results in a small performance hit for the average user. For all but two of the systems monitored, keeping the most active 25% of the files uncompressed and the rest compressed results in less than 5 seconds of CPU time per user per day if the *compress* program is used[3]. Of course, the time spent for decompression will drop as CPU performance continues to increase.

In summary, with a compression factor of 3, a file system that keeps 25% of the data active (uncompressed) uses 1/2 the disk(s) to hold the active 25% of the data and the other 1/2 of the disk(s) to hold the inactive 75% In this case the storage requirements are cut in half. And at the cost of additional decompression cycles, even greater storage savings are possible if less data is kept uncompressed.

[3]At a cost of around 20 seconds per user per day, half of these systems can use the *ppmC* algorithm. (It is our experience that users are willing to tolerate a delay of this length in our environment for example when using the integrated *bags* mail and bboard system.)

4.1 Comparison with standard migration

When migrating files to some backing store, such as a tape archive, performance is impacted more by the number of accesses to the archive than by the amount of data moved. Except for the case of huge files, the migration time is dominated by the access time of the tape, not the amount of data transferred. The time to get a file of size *S* off of a tape archive is computed by a function like *20 seconds + S * 1 second/MByte*. The first term (20 seconds) is the access time, the second the transfer time, which is determined by the transfer rate of 1 second per MByte. On most Unix machines the average file size is on the order of 0.01 MByte, so the average transfer time is approximately 0.01 seconds. Putting one large file onto tape can allow many small files to stay on disk. As a result, it is a good idea to take into account the size of the file when evaluating which files to migrate. The algorithm most commonly used is to migrate the file with the largest *(time since last used) * (size of file)*. This method is called the "*space time product*" algorithm.

For a file system as investigated here, slow storage does not have a large overhead on a per file basis (like the 20 seconds for the above tape). The seek time to slow storage is the same as the seek time to fast storage since both levels are on disk. The transfer rate of the slow storage is significantly worse due to decompression time. Therefore, trading off a higher total data transfer for a reduction in the number of files, as is done in the *space time product* algorithm, would be detrimental in this case.

4.2 Improvements over LRU

The method we used for data collection does not give enough information about individual files to allow evaluation of different migration algorithms. However, for a filesystem as proposed, there are several obvious modifications that can be made to a straight LRU strategy.

Satyanarayanan [16] notes that the difference between time of last read and time of last write (the f-lifetime) can be a useful hint for deciding which files to migrate. A short f-lifetime means a file was only read shortly after it was written, whereas a large f-lifetime means that the file has been read long after it was written. Typically a file that has been read long after it was written has been read many times in between and will be read again. The intuition is that if a file was only read shortly after it was written it is less likely to be read again and therefore is a better candidate for migration.

The above paper also discusses the idea of using filenames as an aid in predicting if a file is going to be read again soon. For example, of the two files "foo" and "foo.BAK", it is more probable that "foo" will be read again. It is common for editors to produce backup files (like "foo.BAK") that are rarely read.

The decision to compress a file should take into account how compressible the file is, as well as its age, since the benefit in terms of disk space will be higher for the files that are highly compressible. For example, if two files have not been accessed in 2 weeks, and one can be compressed to 30% of its original size while the other takes up 90% of its original size after compression, it is clearly more beneficial to migrate the first one. Since the compression is done late at night, it is practical for the system to spend some time deciding what type of compression algorithm to use on each file and how compressible the files are. Similar to the way it is advantageous for systems that migrate files off disk onto tape to be biased towards migrating large files, it is advantageous for a system that compresses files to be biased towards migrating compressible files.

4.3 Prototype implementation

An NFS server was modified to demonstrate the feasibility of including such a file system into an existing operating system. We call this system the ATTIC system because of the similarity between it and the way people put things they have not used recently into their attic, while keeping items they currently use in more accessible parts of their house. ATTIC stands also for "*Active Top Tenth, Inactive Compressed*".

This server is a user level process that accepts NFS requests over the network. This server accesses files through the normal Unix filesystem. The server simply decompresses any compressed file when it is first accessed. Another process, the "migrator", runs late at night and compresses inactive files.

The information indicating whether a file was compressed or not is hidden in the group ID for the file. The process that compresses the file adds 1,000 to the group ID (GID) at compression time. Whenever the server sees a GID over 1,000 it knows to subtract 1,000 from the GID and decompress the file. Except for a slight increase in access time, the fact that some files are compressed is totally invisible to users.

The modifications were straightforward and they resulted in changes to only one part of the fileserver code. All of the changes for decompression were made to the procedure that handles "nfsproc_lookup". To access a file in NFS, the client must first perform a lookup. When the server does a lookup it checks the GID, and decompresses the file if needed. This checking takes almost no extra time because the GID is needed anyway as part of the lookup. Hence, for the uncompressed files, this server is just as fast as the original server.

5 Conclusion

Combining caching and compression has significant advantages over the alternatives of special compression hardware, compressing every file, or relying on users and applications to compress files. Since it requires no special hardware, it does not increase the cost of the system. And unlike a scheme that simply compresses every file, this approach has only a small impact on performance.

The data collected indicate that we can typically expect an ATTIC-type filesystem to double the amount of data that can be stored on a given disk, with only a small performance penalty. For a majority of the systems monitored, our measurements indicate that such a two-level filesystem results in less than an extra 5 or 20 seconds of CPU time per user per day, depending on the compression algorithm used. The real time delay will depend on the load of the system, but is likely to be less than 50 seconds per user per day for systems similar to those that we instrumented, and as processors get faster these times will decrease. In summary, a two-level filesystem that combines caching and compression presents a attractive tradeoff between access speed and increased storage capacity.

Acknowledgments

We thank A. Moffat of the University of Melbourne for the source code for various compression programs. We appreciate the support of S. Ghanem of Hewlett Packard.

References

[1] M. J. Bianchi, J. J. Kato, D. J. Van Maren, "Data Compression in a Half-Inch Reel-to-Reel Tape Drive", Hewlett-Packard Journal, June, 1989, pp. 26-31.

[2] T. Bell, I. H. Witten, J. G. Cleary, "Modeling for text compression", ACM Computing Surveys, Vol 21, 1989, pp. 557-592.

[3] J. G. Cleary, I. H. Witten, "Data Compression Using Adaptive Coding and Partial String Matching", IEEE Transactions on Communications, Vol. 32 No. 4, April 1984, pp. 396-402.

[4] S. Coleman, "Storage in the LLNL Octopus Network: An Overview and Reflections", Sixth IEEE Symposium on Mass Storage Systems, June 1984, pp. 25-30.

[5] G. G. Free, "File Migration in a Unix Environment", Dept of CS, University of Illinois at Urbana-Champaign, Report No. UIUCDCS-R-84-1196, December 1984.

[6] J. H. Howard, M. L. Kazar, S. G. Meenes, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and Performance in a Distributed File System", ACM Trans. Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 51-81.

[7] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", Proceedings of the I.R.E., Vol. 4D, No. 9, September 1952, pp. 1098-1101.

[8] D. H. Lawrie, J. M. Randal, R. R. Barton, "Experiments with Automatic File Migration", Computer, Vol. 15, July 1982, pp. 45-55.

[9] A. Moffat, "Word-based Text Compression", Software-Practice and Experience, Vol. 19(2), February 1989, pp. 185-198.

[10] A. Moffat, "A note on the PPM data compression algorithm", Re. Rept. 88/7, Dept. of Computer Science, Univ Melbourne, Victoria Australia.

[11] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment", Communications of the ACM, Vol. 29, No. 3, March 1986, pp. 184-201.

[12] M. N. Nelson, B. B. Welch, J. K. Ousterhout, "Caching in the Sprite Network File System", ACM Trans. Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 134-154.

[13] D. A. Patterson, G. A. Gibson, R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", ACM SIGMOD88, Chicago, June 1988.

[14] M. Pechura, "File Archival Techniques Using Data Compression", Communications of the ACM, Vol. 25, No. 9, September 1982, pp. 605-609.

[15] P. Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem", Intel Scientific Computers.

[16] M. Satyanarayanan, "A Study of File Sizes and Functional Lifetimes", Proc. Eighth Symp. on Operating Systems Principles, ACM Order No. 534810, December 1981, pp. 96-108.

[17] A. J. Smith, "Long Term File Migration: Development and Evaluation of Algorithms", Communications of the ACM, Vol. 24, No. 8, August 1981, pp. 522-533.

[18] A. J. Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms", IEEE Transaction on Software Engineering, Vol. SE-7, No. 4, July 1981, pp. 403-417.

[19] A. J. Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 94-101.

[20] E. P. Stritter, "File migration", Ph.D. dissertation, Stanford Univ. Comp. Sci. Dep. STAN-CS-77-594, January 1977.

[21] T. A. Welch, "A Technique for High-Performance Data Compression", Computer, June 1984, pp. 8-19.

[22] M. Wells, "File Compression Using Variable Length Encodings", The Computer Journal Vol. 15, No. 4, 1972, pp. 308-313.

[23] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. IT-23, No. 3, May 1977, pp. 337-343.