Two Levels of Filesystem Hierarchy on One Disk

Vince Cate

May 1990
CMU-CS-90-129$_2$

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was supported in part by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, ARPA Order No. 5993 and in part by Hewlett Packard.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

Keywords: filesystem, compression, storage hierarchie
performance measurements, ATTIC, memory hierarchy, fil

Abstract

In many computer systems today, users don't have enough disk space and would welcome a method that made more efficient use of disk. It has long been hoped that compression could be used to increase the amount of user data stored per gigabyte of disk; thus far, attempts to use compression have enjoyed only limited success due to performance degradation, increased hardware cost, or difficulty of implementation.

We present and evaluate a method of using compression that does not suffer from the usual drawbacks. The idea is to have the filesystem separate disk storage into two levels of the memory hierarchy. Files that have not been used recently are compressed, while the other files are kept in standard uncompressed form. In this way, the concepts of caching and compression are used to split a single disk into two levels of storage with different cost and performance characteristics. Inactive files are stored more cheaply, but take longer to access. We call this the ATTIC [1] method.

It is shown that this approach yields an increase in storage capacity nearly as great as that which could be achieved by compressing every file, with nearly the speed of a filesystem that does no compression. Data is presented showing the long term temporal locality of file accesses for 9 systems from 3 different sites. This locality is what makes file migration practical. An analysis of the performance of compression algorithms is presented. Data compressibility and file access patterns vary from machine to machine, but for many systems, the ATTIC approach would result in twice the effective storage capacity while averaging less than 10 seconds of additional delay per user per day.

ATTIC has significant advantages over previous methods. It results in much greater storage capacity than a standard filesystem. It does not require any extra hardware. Once implemented, it can be easily added to existing systems. The compression of files is invisible to users, except for the increased storage and a slight degradation in performance.

1 Introduction

Disk storage costs have been dropping steadily for the last 30 years, but they have continued to be a significant fraction of the cost of a computer system. Many users never seem to have enough disk space. It is well known that much of the data currently stored on disk could be stored in less space if compressed. Several algorithms have been developed in recent years that hold great promise in terms of their speed and how much they compress data [17][9][3]. However, so far, methods for incorporating automatic compression into hardware or the operating system have not been widely used.

1.1 Prior file compression work

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

[1] We've named this idea ATTIC because of the similarity between it and the way people put things they have not used recently into their attic, while keeping items they use in more accessible parts of their house. We could also claim that ATTIC stands for "Active Top Tenth, Inactive Compressed".

1.1.1 Special hardware

One approach is to add special hardware to compress all data before it is stored to disk without degrading performance [18]. Two disadvantages to this approach are that special hardware increases cost, and that it 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.

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 without knowing how much the data will compress. If disks did not need to store fixed-sized blocks of data, or if the disks were to do 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. In [1], special compression hardware is added to the tape drive - resulting in significantly increased tape capacity. In a number of mass storage systems, such as [4], files are compressed when they are moved off of disk onto tape. This approach reduces the number of tapes needed for migration, archive, or backup. While very useful at lower levels of the memory hierarchy, this approach does not improve the disk level.

1.1.2 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 [2] online manual "man" so that it stores its data in a compressed format. This approach has been very useful, but it also has drawbacks. One problem is that it results in significant performance degradation, since every time a file is accessed, it is necessary to decompress it, even when the file is used several times during the same day. Another problem is that it is necessary to modify each application.

1.1.3 Operating system modifications

Another approach is for the operating system to compress all files, as suggested in [11]. 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 much longer to read and decompress a file than to simply read it.

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 very time they are closed and decompressed everytime they are opened.

1.1.4 Manual compression

Another approach, used in [11], is to provide utility programs and let the users compress and decompress their files manually. This approach has been very 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

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

fraction of their total files. Another obstacle keeping users from compressing their files manually is that the applications they use cannot handle compressed files. For example, if a user's mail reading program cannot handle compressed files, the user won't compress his mail files.

1.2 Caching and migration

One of the key concepts in computer science is caching. The goal of caching is to use two types of storage to create a system that has nearly the average speed of a system having only the faster, more expensive storage and, at the same time, nearly the average cost per byte of a system with only the slower cheaper storage. To do this the most active data is kept in a small amount of the fast storage, and the rest of the data is kept in the cheaper storage. If a high percentage of the accesses are to the small amount of active data, the average access time will be near that of the fast memory. Caching with several levels of storage is called a memory hierarchy. A typical memory hierarchy on a small machine has registers, a cache, main memory, and disk. The memory hierarchy of many large systems includes the additional level of automatic tape storage.

We present a concrete example to illustrate the benefits of caching. Imagine a 100 K-byte cache with an access time of 10 ns that costs $\$2,000/MB$ and a 4 M-byte main memory with an access time of 100 ns that costs $\$200/MB$. Let us say that 98% of the time the needed data is in the cache and 2% of the time it is necessary to go to main memory. The average access time will be $0.98 * 10ns + 0.02 * 100ns = 11.8ns$. The total cost of storage is $0.1MB * \$2,000/MB + 4MB * \$200/MB = \$1,000$. Since the cache is replicating data in main memory the total storage the computer sees is only 4 MB, so the average price of storage is $\$1,000/4MB = \$250/MB$. Thus the average access time is not much over the 10 ns of the fast storage, while the average price is not much over the $\$200/MB$ of the cheaper main memory. The same basic concept of caching applies between the other levels of the memory hierarchy.

When caching between disk storage and tape storage, a file is usually removed from one media when written to the other. This is different from a normal cache where data is not removed from the larger storage when copied to the faster storage. Because files are moved back and forth, this type of caching is called *file migration.*

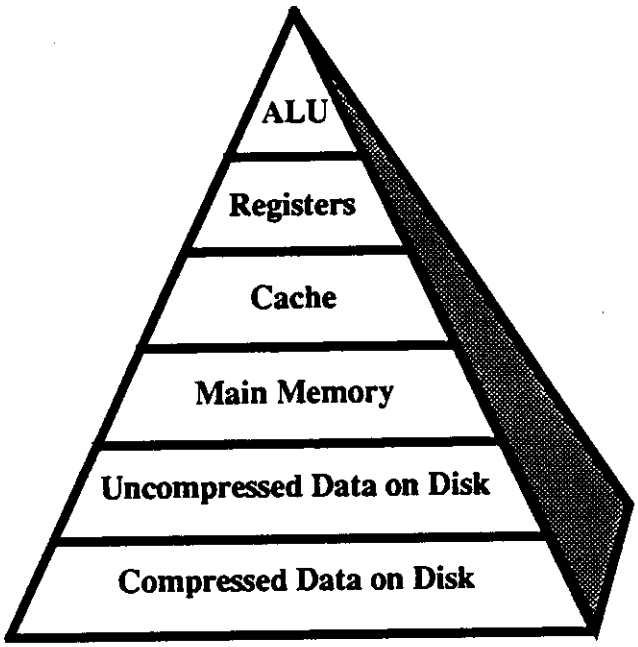
1.3 Combination of compression and migration

The ATTIC method combines the ideas of compression and migration. 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 costs less in terms of disk space to store, but has slower access times. Files migrate between these two levels in much the same way that files migrate between disk and tape. Figure 1 shows the new memory hierarchy.

In general, the least recently used files are considered "inactive" and compressed. The most recently used files are called "active" and kept uncompressed. Every night some of the active files are reclassified as inactive and compressed. Which files to reclassify will be based mainly on the time the file was last accessed. A modified version of the LRU (Least Recently Used) algorithm is proposed later in this paper as a more effective method for deciding which files to compress.

When a compressed file is opened, the filesystem automatically decompresses it before the open completes. Thus, users and applications need not be aware of which files are compressed and

Figure 1: The ATTIC Memory Hierarchy



which ones are not. No applications will need modification, nor will users be required to compress files.

Since only inactive files are compressed, decompression does not occur every time a file is accessed. As will be shown later, typical file access patterns are such that even if 90% of the files on a disk are compressed, only a relatively small number of the files accessed on a given day will need to be decompressed.

Backups can be done in such a way that compressed files will not be decompressed when stored to tape. For a given amount of user data, the time to do backups and the number of backup tapes will both be reduced. The compression, therefore, saves both disk space and tape space. Again, these benefits are realized without any special hardware.

If disk space runs out during the day, the process that compresses files can be run to create more space. If this happens often the system needs more disks. Note that with this design, it is possible for a user to get the error message "filesystem full, pausing" when he reads an inactive file if the disk is full. This is because the filesystem will try to decompress the file, and to do so, it will need more disk space. However, unlike current filesystems, an ATTIC filesystem will be able to take action to create more space.

There are three main factors that determine how well this filesystem design will work: how much temporal locality there is in file access patterns, how compressible files are, and how fast files can be decompressed. compression time is not critical because, on a system with sufficient disk, it is only done late at night. These three factors are addressed in the following sections of this paper.

2 Long term temporal locality

To evaluate the overall performance of an ATTIC type filesystem, we need to know how often accesses will come out of the fast storage and how often it will be necessary to go to slow storage. We are therefore led to ask: what is the long term temporal locality of file access patterns? This same locality makes file migration possible, so the migration literature is relevant to this investigation.

2.1 Prior migration work

Many studies have been done relevant to file migration [16][13][14][12][7]. Although the locality depends on the workload, the same trace was used in three of these papers [16][13][14]. This trace was obtained by instrumenting an editor at the Stanford Linear Accelerator Center. The trace covers about a year, starting in August 1974. The users on this system had a small amount of disk space (70KB to 1.4 MB), and they manually migrated their files to tape.

As was later confirmed in our studies, we expected the locality in current systems (1989) to be much greater. Our logic was as follows. Today it is much less common for users to move files off disk and onto tape. If more inactive data is left on disk, then a smaller portion of the data on disk will be active. If only a small portion of its data is active, a system has has good temporal locality. Thus we collected data on the long term file access patterns from a variety of current systems. During the period traced, starting in 1974, the SLAC system accessed about 40% of its disk storage on an average weekday. Many of the systems we monitored accessed less than 4% of the available disk storage on the average day.

Using simple trace-driven simulations to determine the level of long term temporal locality in file access patterns, it would be necessary to monitor a system for a very long time, such as a year [16]. Also, the data from tracing all accesses to files is extremely large. Even though there are methods for reducing the size of traces [15], the amount of data is still cumbersome.

2.2 Data collection method

We have an easy way to collect manageable amounts of data that indicate how well an LRU migration/caching algorithm would work on a given system. Unlike other methods, our method does not require any kernel modifications. 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 would have performed on that day for cache sizes corresponding to the amount of data accessed in the last 24 hours, 48 hours, etc.

We know that a real implementation can do at least this well, since the LRU algorithm is easy to implement. Therefore, this data gives an upper bound on the miss rate.

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 things such as the number of files of each age and the total number of bytes for files of each age.

Figure 2 demonstrates how the number of files accessed from each age of file is calculated from the histogram data. For example, at the start of day 1, there were 1,000 files that were last read between 24 and 48 hours ago. At the start of day 2, there were 700 files that were last accessed

Age	Start of Day 1 Files	Start of Day 2 Files	Number 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

Figure 2: Using Histograms to Calculate Number of Files Accessed

Age	Start of Day 1 K Bytes	Start of Day 2 K Bytes	K Bytes 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

Figure 3: Using Histograms to Calculate Total Bytes 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 figure 3 was generated in this way. At the start of day 1, there were 764 K bytes of data were last accessed between 8 and 9 days ago. At the start of day 2, there were only 701 K bytes of data accessed between 9 and 10 days ago. This indicates that 63 K bytes of the original 764 K bytes of data that started day 1 at this age were accessed during day 1.

The right column in figure 3 shows the amount of data accessed for each age. From this we calculate the size of a cache necessary to hold all of the data accessed in the last N days by adding up the first N numbers in the right column. We can then tell how much data would not have been in a cache of that size by adding up all of the numbers below row N in the 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.

2.3 Data collection implementation

Programs were written to collect data on both System V and 4.2 BSD versions of Unix. The monitoring program makes one pass over the inodes to collect the data. This tends to take about 5 minutes per gigabyte of disk. Since this is done late at night, monitoring a system does not impact its performance significantly.

As opposed to trace data for the filesystem, this turns out to be a manageable (small) amount of data. We can therefore send the data collected from remote machines via email. 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.

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. It is therefore 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 9 systems.

This data collection method requires a "time of last access" for each file. The preferred method is to take the maximum of the ATime, MTime, and CTime for a file; these are the time of last read, time of last write, and time of last inode modification. However, on some systems, the CTime was not used in this calculation because it was artificially updated by the system backup process. The reason for wanting to include the CTime is that it is possible to copy a file with *"cp -p old new"* such that the read and write dates on the new file are the same as those on the old, but the CTime will be correct. The difference between just using the ATime and using both the ATime and MTime is due to files that are written without being read. This was noticeable but small.

2.4 Limits on data collected

One problem with this method of data collection is that it does not indicate how well caching would work 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 would be 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 the ATTIC approach would work for larger cache sizes. For many of the systems measured, around 5% of the total storage space was accessed on a given day.

Studies using traces with only one bit of information per file per day will have this same problem. With a trace resolution of one day, it is not possible to accurately predict the miss rate for a cache smaller than the amount of data accessed in one day. It is interesting to note that in the trace used in [16][13][14], around 40% of the data on disk was accessed on the average working day. Thus in the analysis in [13], cache sizes had to be large enough to hold at least 14,000 out of the 33,000 total blocks on disk.

Another limitation of our data is that a file deletion is counted as a read. In the previous example, on day 1 there were 1,000 files between 24 and 48 hours old and then on day 2 there were 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 is 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 there was less than a 1% increase in disk usage during the day for all systems and all days. It will be seen later that this level of precision is sufficient to justify the conclusions of this investigation.

2.5 Difference from 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 off of a tape archive will be a function like *20 seconds + 1 second/MB*. The 20 seconds is the access time and the 1 second/MB is the transfer rate. On most Unix machines the average file size is on the order of 0.01 MB, so the average transfer time would be around 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 an ATTIC filesystem, 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 will not be much different than the seek time to fast storage since they are both on disk. Assuming a single compression algorithm, the decompression time for a file is a function its size and its compression ratio. In general, more compressed files will require less disk and CPU time per decompressed byte of user data. If all files had the same compression ratio, the decompression time would scale linearly with the size of the file. For example, a machine might decompress data at a rate of *10 seconds/MB*. The extra delay in accessing a compressed file is simply a function of the length of the file. In this case there is no motivation to reduce the number of files accessed from slow storage, just the amount of data accessed. The space time algorithm, on the other hand, reduces the number of files transferred at the cost of increasing the amount of data transferred. For an ATTIC type system, this is a detrimental tradeoff.

2.6 Improvements on LRU

In contrast to [13], this paper cannot compare migration algorithms. The method we used for data collection does not give enough information about individual files to allow evaluation of different migration algorithms. However, for an ATTIC filesystem, there are several obvious improvements that can be made to a straight LRU strategy.

Satyanarayanan [12] notes that the difference between time of last read and time of last write (the f-lifetime) can be a very useful piece of information for deciding which files to migrate. For example, assume there are two files, A and B, that were both last read a week ago, and also assume that file A was last written a year ago while file B was last written a week ago. The f-lifetime of B is zero and the f-lifetime of A is 358 days. There is a good chance that A has been read many times since it was last written and that it will be read again. The odds of A being used in the next

week are higher than the odds of B being used again in the next week.

Satyanarayanan 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. There are two reasons for this. First, files that are very compressible take less time to decompress. Second, the benefit in terms of disk space will be higher for the files that are very 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 migrates 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.

3 Fast storage

3.1 Cost

The uncompressed files take up as much disk as they do with the current operating system. This means that the cost per megabyte is exactly the same in the current filesystems as it is for the "fast storage" in the proposed system.

3.2 Speed

The uncompressed files can be accessed just as fast as they can be accessed in current systems. The only extra work that the filesystem must do is to check that the file does not need to be decompressed before it accesses it; this check takes little time to perform.

3.3 Potential improvements in speed

Even a small improvement in the time to access active files can compensate for the significantly slower access time of the low-cost storage, since the vast majority of the accesses will be to the active files. Our two level disk hierarchy makes such improvements natural. If the average disk access time of a system were reduced from 30 ms to 20 ms on a system that was doing 30 disk I/Os per second, it would save the users of this system 0.3 seconds of time every second. Over an eight hour day, this is more than two hours saved, easily making up for the few extra minutes per day spent on decompression.

3.3.1 Reducing seek times

If all of the active files are put in the same part of the disk, then the average head seek distance is reduced. If the active files are on 10% of the disk, then most of the time the head will be seeking

within this 10% of the disk. For random seeks, this would cause the average seek distance to be 1/10th as much. For a typical disk drive, the seek time is composed of a fixed settling time plus a time proportional to the square root of the number of tracks covered. For a typical drive, random seeks of 1/10th the distance result in about 1/2 the average seek time, not 1/10th as one might expect.

During the night, a program could move files around on the disk so that the recently used files were near the center tracks. It could also move files around to place the free sectors in the center tracks so that files created on the next day would also be in the center.

New optical disks have very fast seek times for nearby sectors. Putting the active files in the same part of the disk could have an even more dramatic impact on the average seek time for an optical drive.

3.3.2 Reducing number of seeks

If the active files are moved around during the night to reduce the average seek distance, we can reduce the number of seeks by also expending some extra effort to store each file contiguously on the disk. Since the active files are a small fraction of the total files, moving them around once a night is manageable. The risk of data loss can be minimized by moving the files around after the nightly backup is done. Every weekend the whole disk could be repacked.

3.3.3 Load balancing for speed

Currently it is common for one disk to run much busier than others attached to the same computer [8]. Thus, the busy drive frequently has multiple I/O requests outstanding while other drives are idle. If the active files could be evenly spread over the various drives, the average I/O request queue would be shorter and the I/O response times would be better.

In most current Unix systems it is possible to look at the first part of a file's full path-name and determine which disk the file is stored on. It would require a major modification to such a filesystem to load balance the disks. In some systems, such as Andrew [10], a file's name does not indicate which server it is on, let alone which disk. In Andrew, operators can, and do, move files from one server to another without users ever knowing about the switch. This could be automated and made to balance the load across disks as well as servers.

4 Low cost storage

By compressing files, more bytes of user data can be stored on a given dollar's worth of disk. Therefore, a filesystem that compresses files will store data more cheaply. The drawback is that access to compressed files will be slower since it takes time to decompress them. We would like to be able to answer these two questions about our low-cost storage, "How much cheaper is it?", and "How much slower is it?". The answers to these questions depend on the compression algorithm and processor used.

4.1 Cost savings

The cost savings factor is nearly equal to the compression ratio; if data can be compressed by a factor of 2 the cost of file storage can be reduced by about a factor of 2. The reason the compression ratio does not *exactly* equal the cost savings factor is that the internal fragmentation[3] will be different.

There are specialized algorithms for compressing pictures[2] and others particularly suited to English text [9][3]; there are also very fast general purpose algorithms [19][17]. The special-purpose algorithms can achieve very high compression ratios on the type of data they are designed for. On English text, Moffat [9] reports compression ratios of 3.5. For a C program, Moffat reports nearly a factor of 5 compression.

Welch [17] describes a very fast implementation of the Lempel Ziv algorithm [19]. Welch's paper lead to the Unix "compress" program. This program is fast and does a good job of compressing a wide range of files. For English text, the *compress* program gets about a factor of 2, and on C programs, it gets about 2.5. The compression ratio on executable binaries depends on the instruction set. For VAX instructions, *compress* will reduce a file to 2/3 its original size. For RISC instruction sets such as SPARC or HP-PRECISION, it achieves about a factor of 2. In two environments characterized by program development, text processing, and email, an average compression factor of nearly 2 has been observed using a program that traversed a directory tree and compressed every file using the compress program.

In the proposed filesystem, the compression can be done at night when there is plenty of idle CPU time. Several different algorithms can be tried on each file to see which one works best. Thus, it should be possible to use a special purpose algorithm when appropriate; the general purpose algorithm will only be used when there is not a better special purpose algorithm to use on a file. Thus we can expect the average compression for many environments to be between a factor of 2 and 3.5.

Special purpose algorithms can be added to the system. One special purpose compression algorithm that should improve storage is one that can recognize C runtime libraries when it comes across them in executable files. For this to work, the algorithm must adjust for the absolute addresses which change from binary to binary. If this were done, it should be possible to encode a large library routine in just a few bytes.

4.2 Performance degradation

When a file is read from the low cost storage it will need to be decompressed. This will result in extra CPU usage and in most cases slower real access times. The decompression time will depend on the CPU speed, memory bandwidth, current load, and compression algorithm used.

As a data point, Lempel Ziv takes about 5 seconds of CPU time per decompressed megabyte on an HP 835 or a Sun 4. On a Sun 3/50, it takes about 15 seconds. A microvax takes about 26 seconds. Processors continue to get faster, so this will be reduced as time goes on. In some cases, there will be a choice between a faster algorithm and one that compresses the data more. One possibility is to use a fast algorithm like Lempel Ziv on files that are just old enough to warrant compressing but not very old and an algorithm like [9], which compresses files more but is estimated to take twice as long, for the older files.

[3] Internal fragmentation is the amount of wasted disk space due to the allocation of fixed size pieces of disk to files.

In this study we ignore the I/O time saved by reading compressed files. Given a fast enough CPU, it would be possible for a system to achieve higher bandwidth when reading compressed files than when reading uncompressed files. However, this is unlikely for at least the next 5 years.

4.3 Improvements when used in conjunction with migration

A compressing filesystem can be used in conjunction with standard file migration. The increase in effective disk storage space will decrease the number of accesses to the next level in the memory hierarchy (tape). The time to decompress the average file will be much less than the time it would have taken to get the file off of tape. Files migrated to lower levels in the hierarchy will already be compressed and so take up less storage space.

5 Results

The measured locality and compression algorithm performance combine to indicate that an ATTIC type filesystem would work well on many systems.

For this study, the miss ratio is not as important as the absolute amount of time that users are delayed. To get the absolute delay, we 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 a delay per user. We can then decide if the performance degradation would be acceptable to users.

The attached graphs show how many kilobytes a system would have to decompress per day as a function of the percentage of the data kept uncompressed. All graphs are on a log-log scale. Also attached are graphs showing how much time per user this decompression would take. The program that monitors a system keeps track of how many different user IDs are seen for the active files. This will catch every user who logs in anytime during the day since his .login or .profile file will be read. This also counts references to files owned by system accounts, such as "mail" or "root", as a login; therefore, the number of system accounts was manually subtracted from the measured number of users.

Since decompression is compute bound it will be done at a lower priority than interactive jobs due to the way schedulers work. Hence, other users on the system will not be affected much when one user accesses a compressed file. The general increase in CPU load will not be noticed for systems with reasonable loads. For the systems monitored, this increase would be less than 5 minutes of CPU time over the whole day. Since the individual users will be waiting, the CPU time per user is plotted. 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. Most general purpose systems seem to be less than 50% utilized. The CMU machines take about 26 seconds to decompress 1 megabyte of data, and the other machines take 5 seconds per megabyte.

Table 5 describes the systems monitored; they are all Unix systems. The three systems at CMU are used by computer science graduate students. The other systems are used for various types of research and development.

The most peculiar system we monitored is hpcupt1. This system is used almost exclusively for reading netnews. On this system, most users do not log in for very long, so the average number of users logged in at once is much less than the average number of users in a day. Netnews files are

Location	Machine	Total Disk In MBytes	Avg Users In One Day	Days Monitored	Secs of delay/User/Day at 25% uncompressed
ATT	corona	3,800	71	57	2.5
HP	hpcupt1	917	110	78	0.3
HP	hpdblpy	1,920	22	38	15
HP	hpperf1	2,210	61	64	1.5
HP	hpisoi0	1,440	35	16	10
HP	hpisoi2	550	9	20	1
CMU	sam	690	25	230	1
CMU	unh	521	8	278	3
CMU	b	952	43	282	3

Figure 4: Systems Monitored

read a lot when they are new, and our monitoring program records them as being read when they are deleted. This peculiar usage explains the two "bumps" in the graphs for hpcupt1.

The last column of table 5 is taken from the graphs. It shows how much extra CPU time would be needed for decompression for each user each day if all but the most active 25% of the data were compressed. As is evident from the data presented, how well the ATTIC approach will work varies from system to system. It should be noted that the ATTIC approach will not work on a system that uses a small number of large files. In particular, a system that uses one large data-base file will not gain anything from this approach. The hpdblpy system is in the data-base lab at HP 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 an ATTIC type of file system.

For all but two of the systems monitored, keeping the most active 25% of the files uncompressed and the rest compressed would result in less than than 4 seconds of CPU time per user per day.

If only 25% of the disk holds uncompressed files then the rest of the disk can be used to hold compressed data. If 75% of the disk were used to hold compressed data at an average compression factor of 2, the users would be able to store 75% more data on disk. If an ATTIC filesystem made 1 gigabyte of disks (say $5,000) seem like 1.75 GB, it would be like giving the users $3,750 worth of disk for free, except for the slight performance degredation.

We feel that the vast majority of systems will realize better than a 75% increase in storage space. On a RISC architecture, binaries are very compressible, and overall system compression ratios will generally average well above 2. Furthermore, we can keep less than 25% of the data uncompressed on many systems. Thus, while some systems will get a little less than a factor of 2 overall increase in storage, many systems will more than double their effective storage space using the ATTIC approach.

6 Prototype implementation

A NFS server was modified to demonstrate the ATTIC concept. This server is a user level process that accepts NFS requests over the network. The server accesses files through the normal Unix filesystem. This made it easy to modify.

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 the performance, 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 major changes for decompression were made to the procedure that handles "nfsproc_lookup". In order to access a file in NFS, the client must first do 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.

7 Conclusion

The data collected indicate that the ATTIC filesystem design would typically double the amount of data that can be stored on a given disk with a very small performance penalty. For most systems monitored, measurements indicate that it would result in less than an extra 5 seconds of CPU time per user per day. The real time delay will depend on the load of the system, but should be less than 10 seconds per user per day for most systems. As processors get faster these times will decrease. Moreover, if a couple of additional modifications are implemented at the same time, it should be possible to create a filesystem that is faster and has more than twice the storage capacity.

This approach has significant advantages over the alternatives of special compression hardware, compressing every file, or relying on users and applications compress files. Since it requires no special hardware, it will not increase the cost of the system. Unlike the scheme that simply compresses every file, this approach has only a small impact on performance. Users do not, in practice, compress a very large percentage of their files, so much more space is made available by the proposed system.

In conclusion, the ATTIC approach looks very promising.

References

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

[2] CCITT Red Book, Vol. VII.3, recommendations T.4 and T.6, 1984.

[3] John G. Cleary, Ian 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] Sam Coleman, "Storage in the LLNL Octopus Network: An Overview and Reflections", Sixth IEEE Symposium on Mass Storage Systems, June 1984, pp. 25-30.

[5] Gordon George 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] David 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.

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

[8] Joe Martinka, personal communication.

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

[10] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment", Communications of the ACM, Vol. 29, No. 3, March 1986, pp. 184-201.

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

[12] 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.

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

[14] Alan Jay 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.

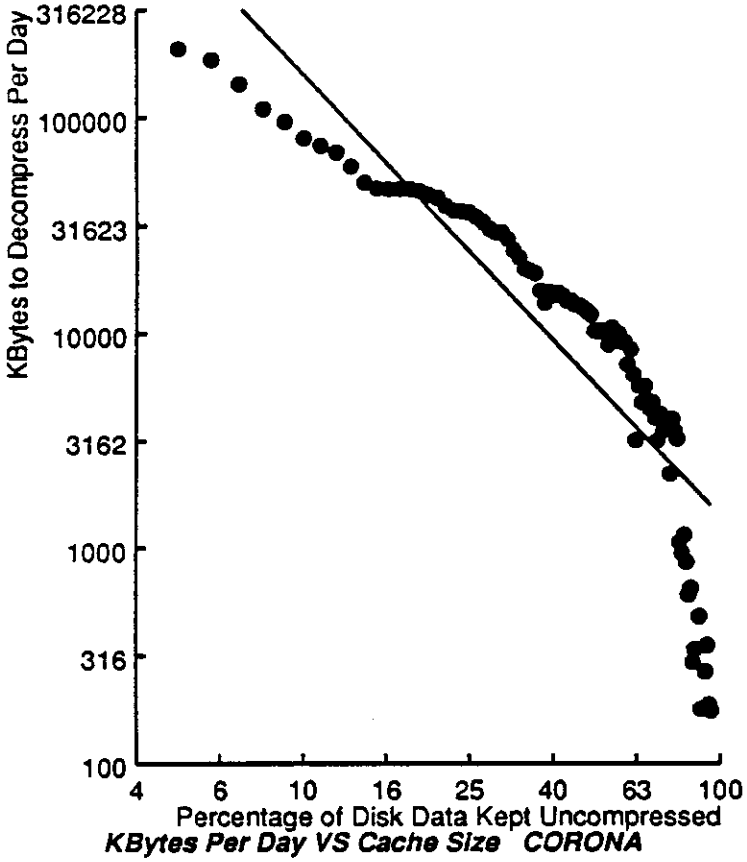
[15] Alan Jay Smith, "Two Methods for the Efficient Analysis of Memory Address Trace Data", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 94-101.

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

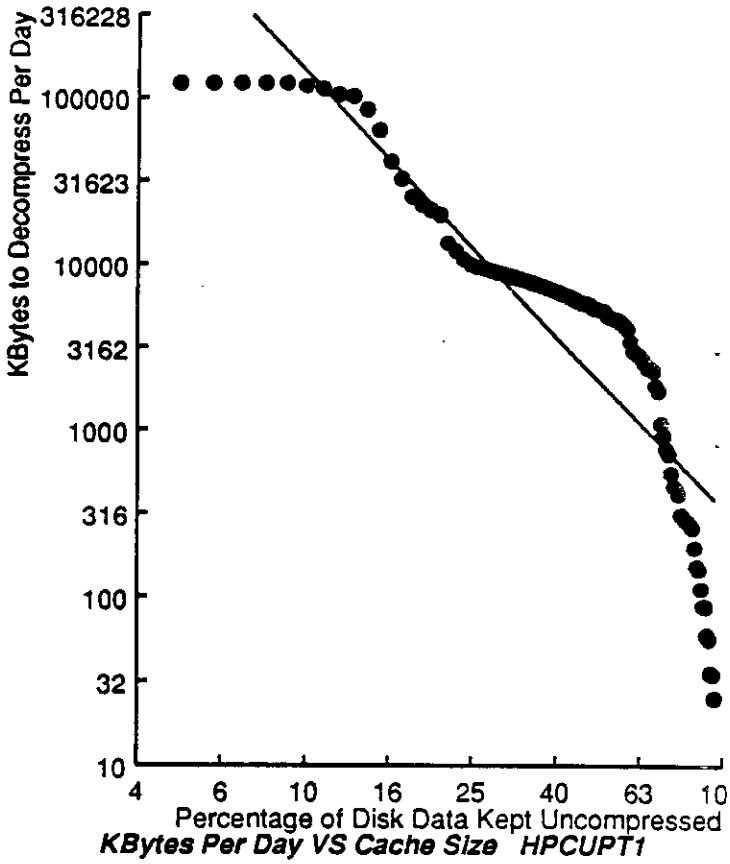
[17] Terry A. Welch, "A Technique for High-Performance Data Compression", Computer, June 1984, pp. 8-19.

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

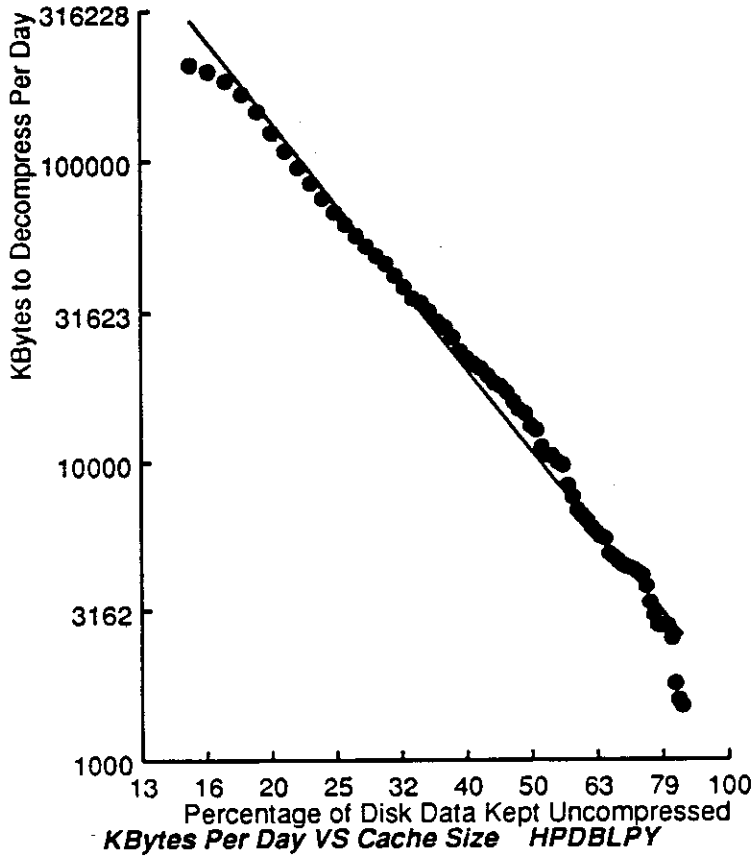
[19] Jacob Ziv, Abraham Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. IT-23, No. 3, May 1977, pp. 337-343.



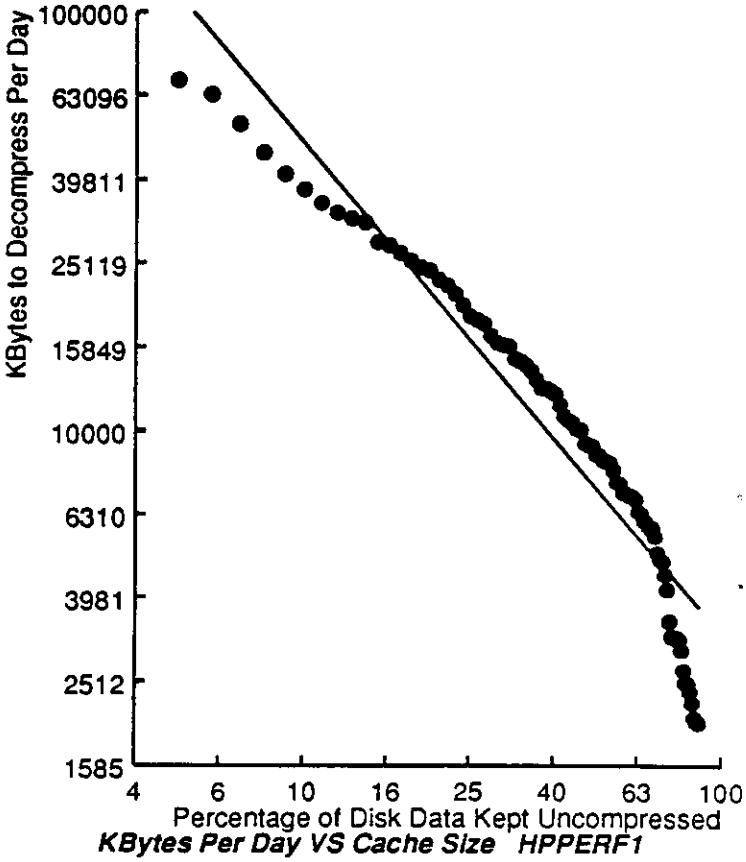
KBytes Per Day VS Cache Size CORONA



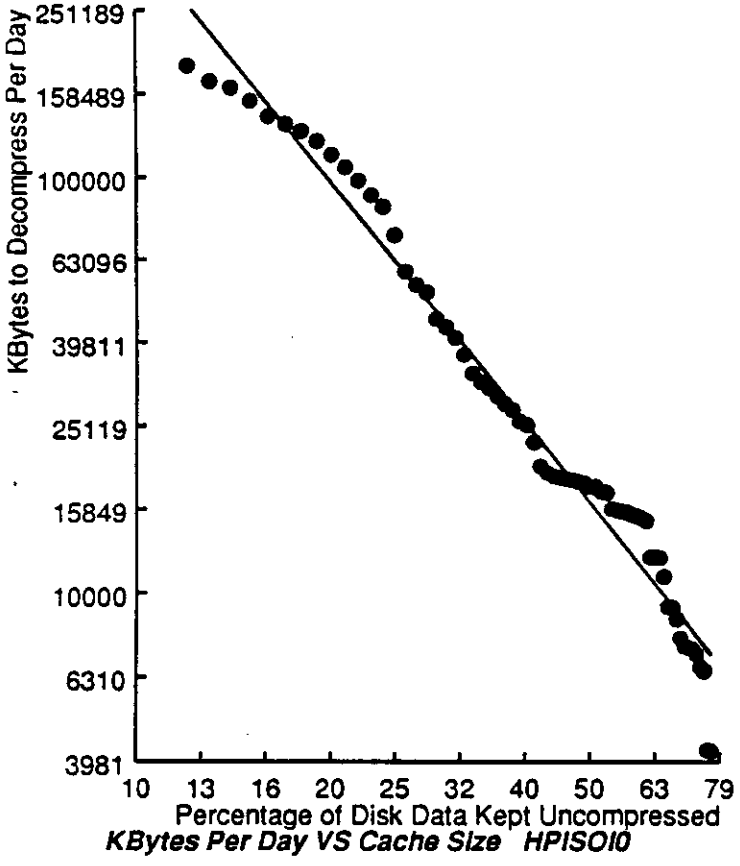
KBytes Per Day VS Cache Size HPCUPT1



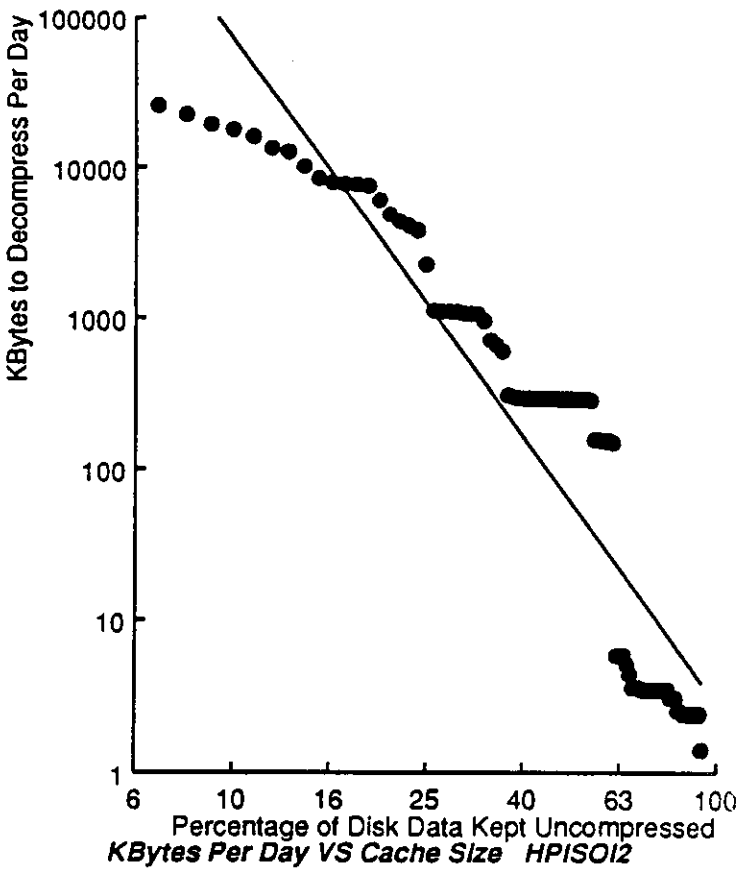
KBytes Per Day VS Cache Size HPDBLPY



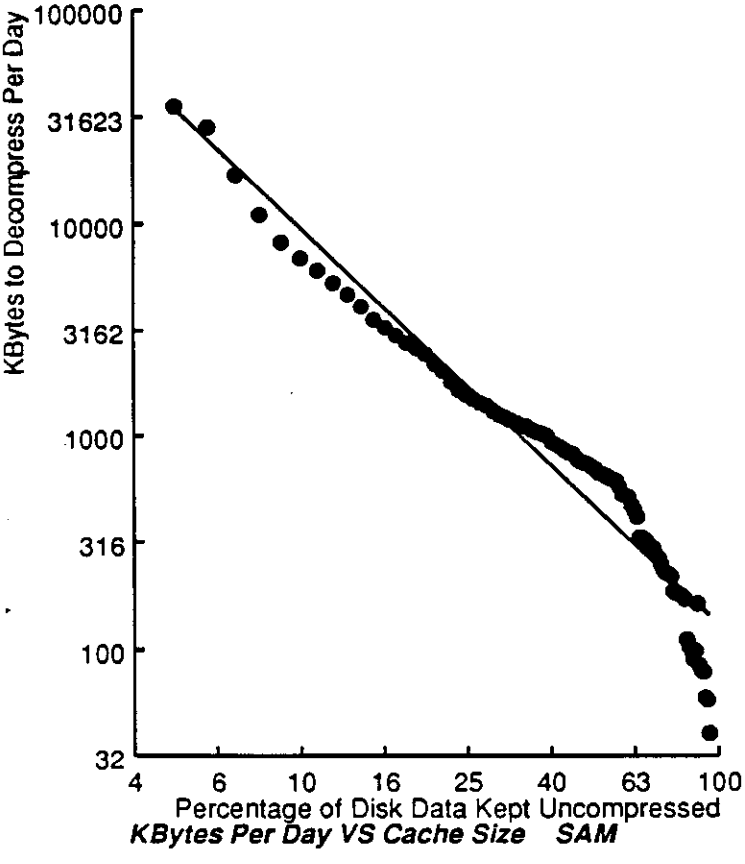
KBytes Per Day VS Cache Size HPPERF1



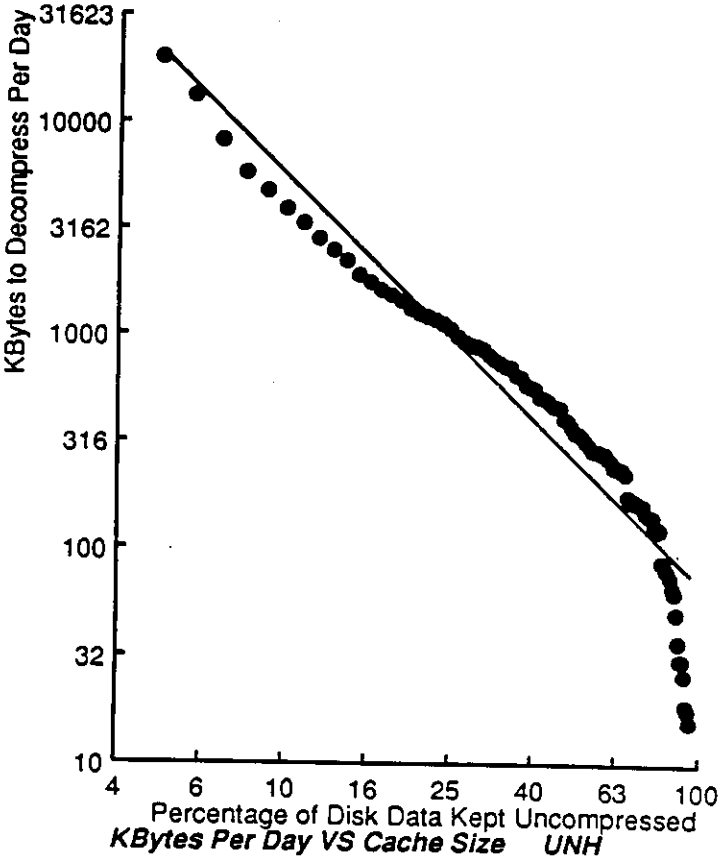
KBytes Per Day VS Cache Size HPISOI0



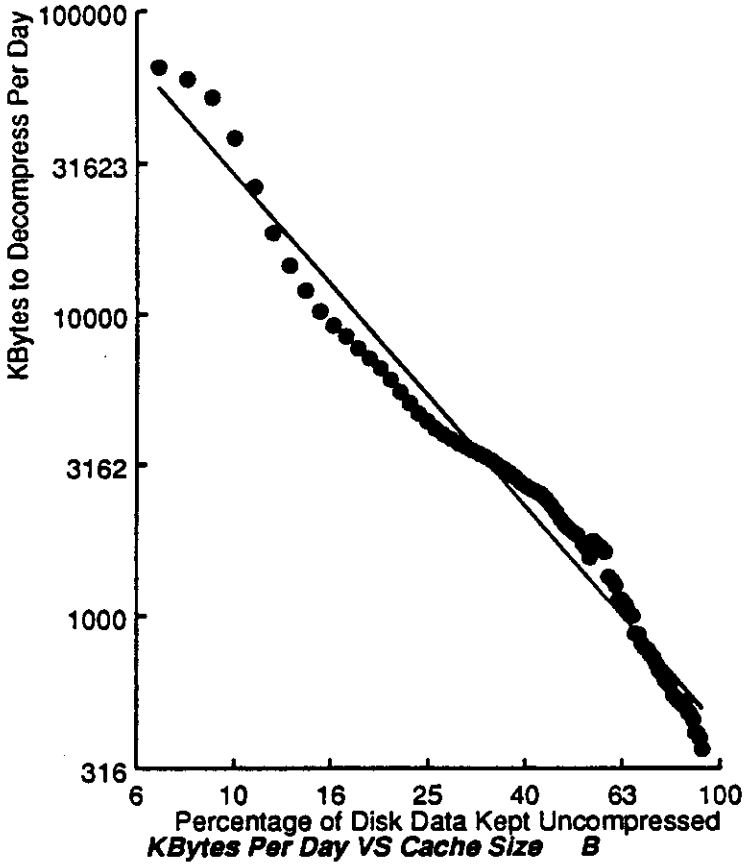
KBytes Per Day VS Cache Size HPISOI2



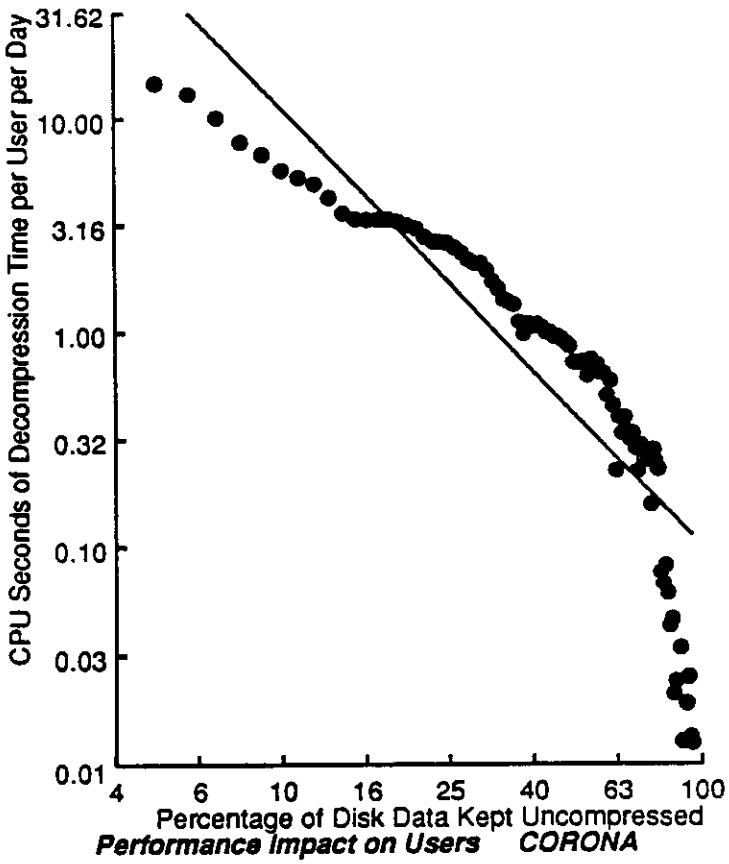
KBytes Per Day VS Cache Size SAM



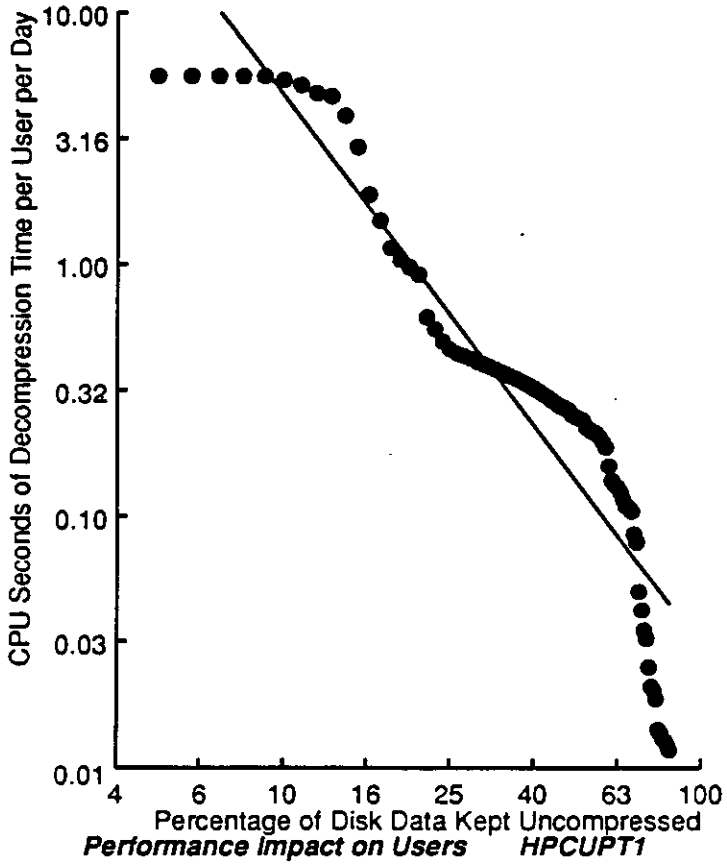
KBytes Per Day VS Cache Size UNH



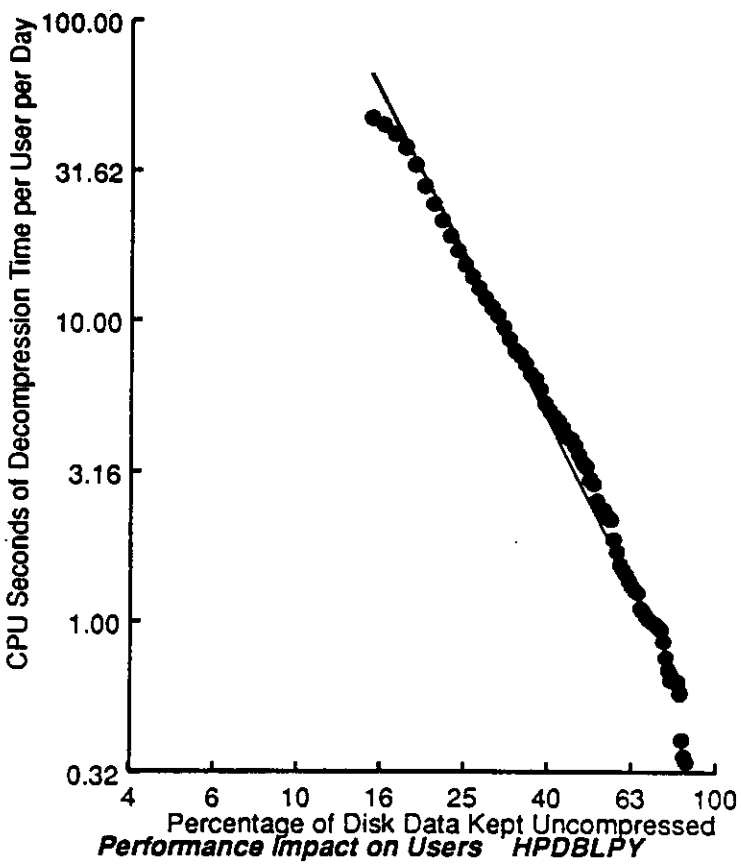
KBytes Per Day VS Cache Size B



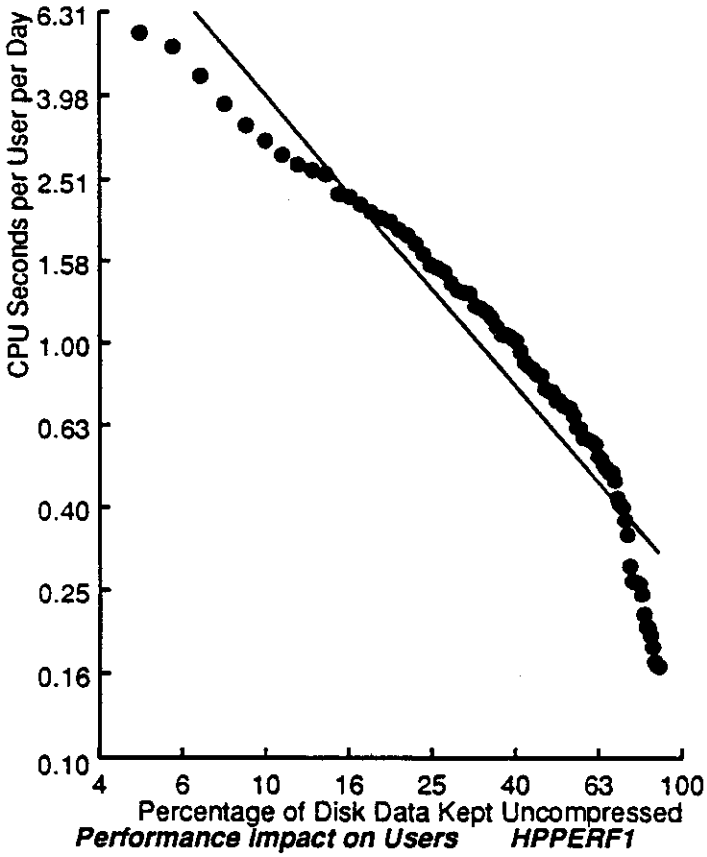
Performance Impact on Users CORONA

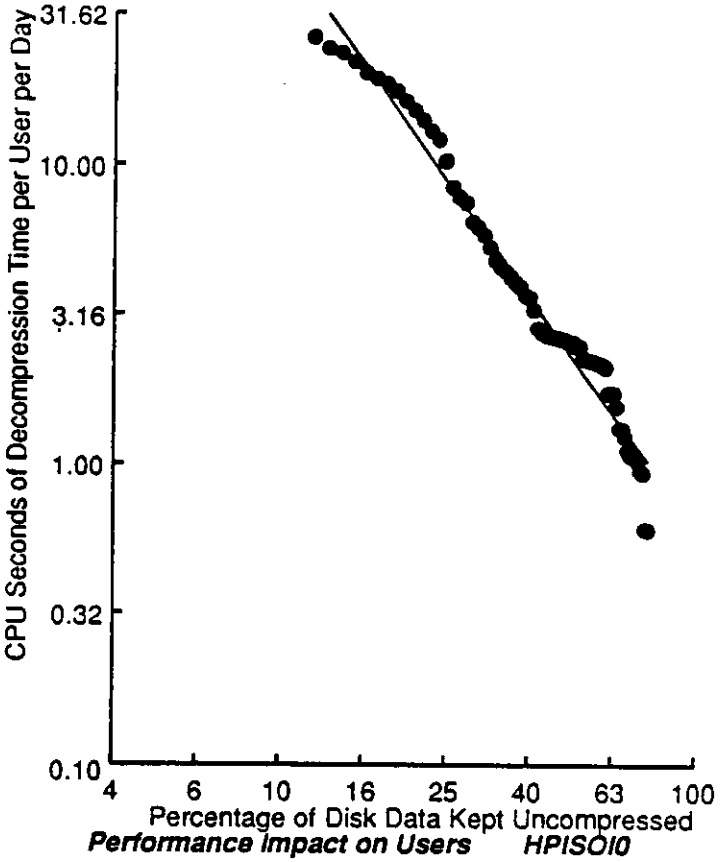


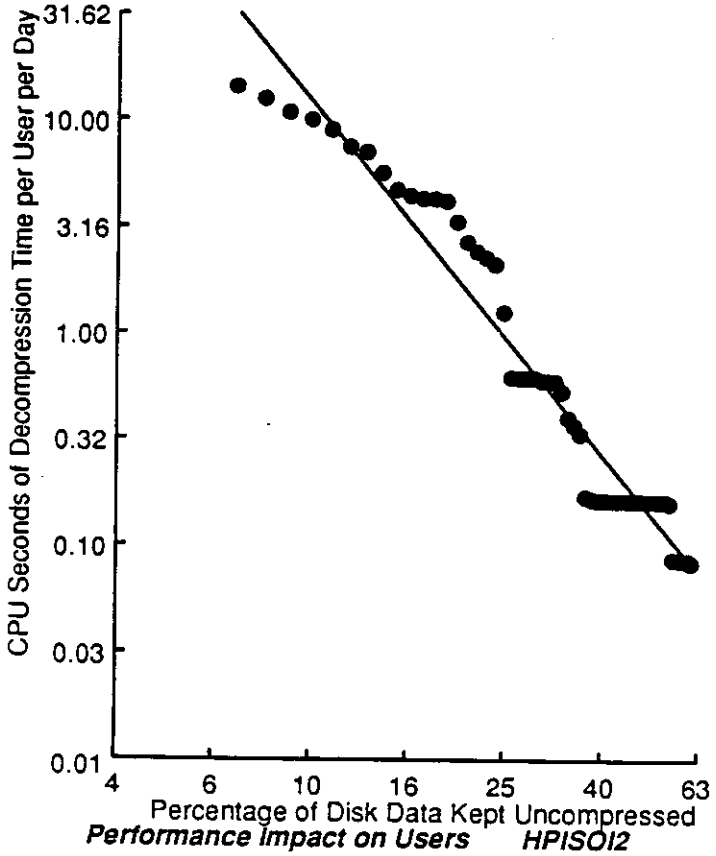
Performance Impact on Users HPCUPT1

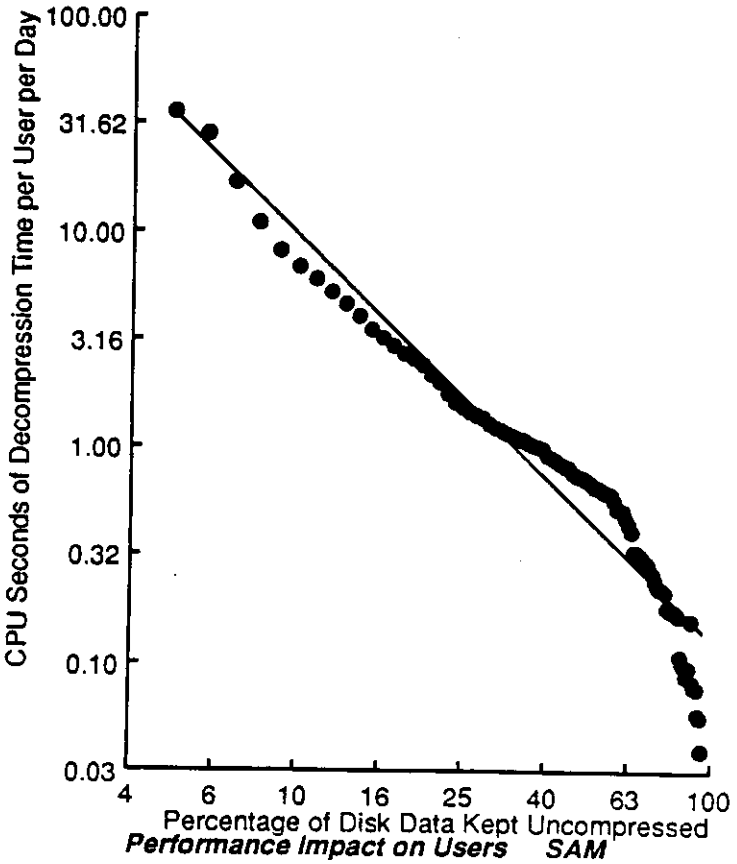


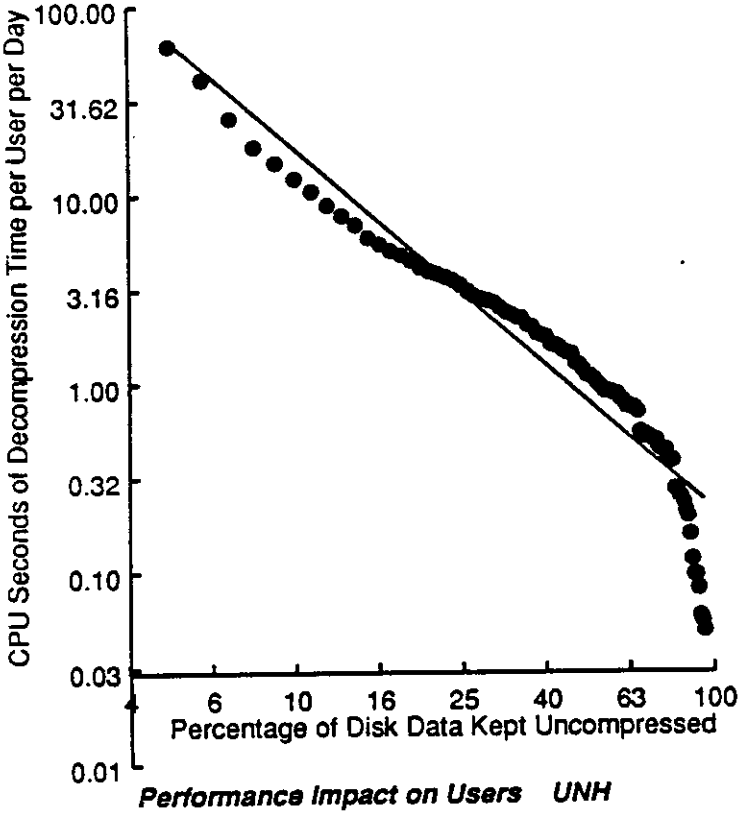
Performance Impact on Users HPDBLPY


CPU Seconds per User per Day
6.31
3.98
2.51
1.58
1.00
0.63
0.40
0.25
0.16
0.10
4
6
10
16
25
40
63
100
Percentage of Disk Data Kept Uncompressed
Performance Impact on Users HPPERF1

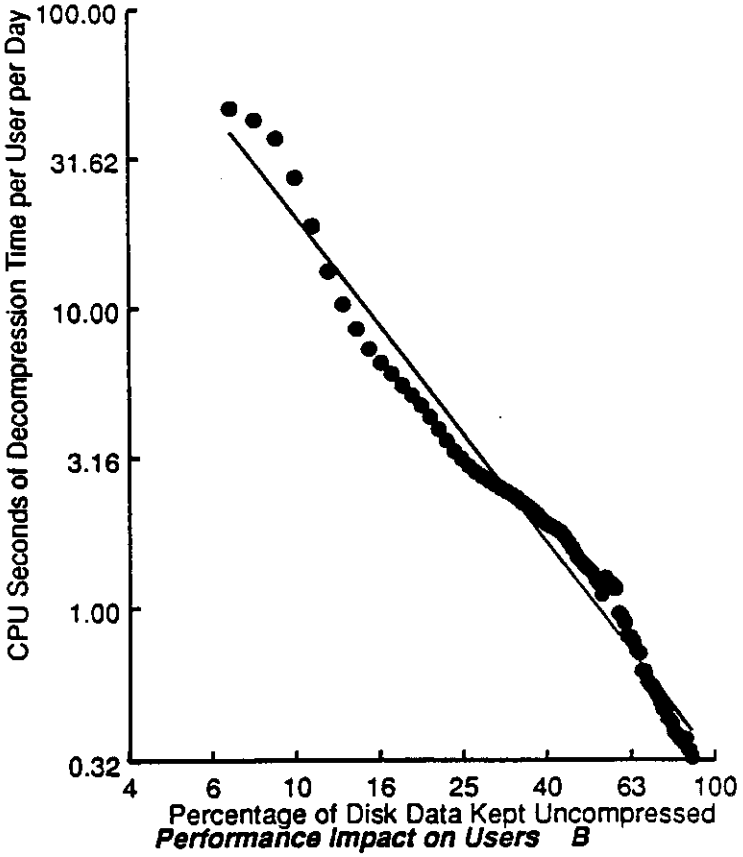

CPU Seconds of Decompression Time per User per Day
31.62
10.00
3.16
1.00
0.32
0.10
4
6
10
16
25
40
63
100
Percentage of Disk Data Kept Uncompressed
Performance Impact on Users HPISOI0


CPU Seconds of Decompression Time per User per Day
31.62
10.00
3.16
1.00
0.32
0.10
0.03
0.01
4
6
10
16
25
40
63
Percentage of Disk Data Kept Uncompressed
Performance Impact on Users HPISOI2


CPU Seconds of Decompression Time per User per Day
100.00
31.62
10.00
3.16
1.00
0.32
0.10
0.03
4
6
10
16
25
40
63
100
Percentage of Disk Data Kept Uncompressed
Performance Impact on Users SAM



Performance Impact on Users UNH



Performance Impact on Users B