A Fast File System for Caching Web Objects

Matthew McCormick Jonathan Ledlie
{mattmcc, ledlie}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, Wisconsin 53706, U.S.A.

Abstract

Given the increasing performance of network connections and the slow nature of disk drives, the authors propose a new file system for web proxy servers that cache files on a local disk. This file system, referred to as the cache file system (CFS), utilizes the facts that cached web pages do not change in size once they are cached, do not have changing permissions, and do have a back-up version at the original server. Immutable files allow all data in a file to be written in contiguous blocks on disk. Reading and writing to contiguous blocks allows the best utilization of a disk's bandwidth and results in the lowest possible seek times. The fact that all data is backed up on remote disks allows CFS to store all meta information in memory and not checkpoint. Should the system crash and this data be lost, users will still get the correct data – it will just be coming from the remote server until the cache is reloaded. By taking advantage of these invariants, CFS is able to out perform a Unix file system by at least 50% on several benchmarks.

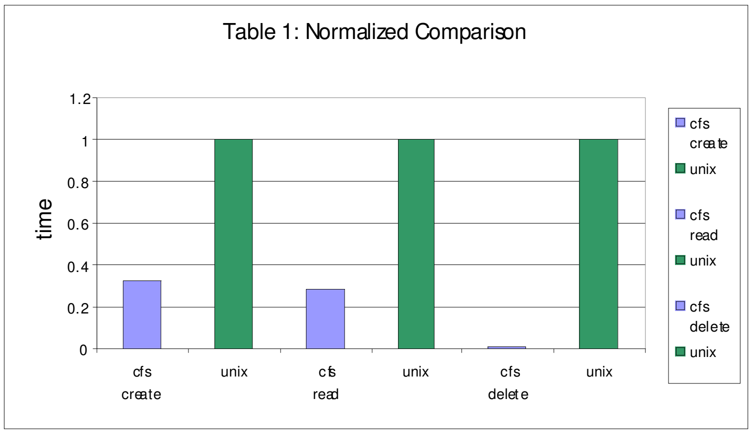
1 Introduction

Cacheable internet files embody certain characteristics which enable – and demand – rapid access. Some web files, like search queries are not easily cacheable without knowing the query's schema. Others, like relatively static images and text files, are, and need to be, cached to reduce both local and server-side workloads. These files may have owners back at their servers, but, once out, they are anonymous, non-volatile, and easily recoverable. These cacheable web files only consist of a URL, some data, and an expiration time. Their most important requirement is quick transfer to the client. Storing these files in the UNIX file system gives them more generality than they need and this flexibility makes their access slower than it should be. Other file system work has shown that by taking advantage of invariants, a much faster solution is often possible.[1] By optimizing on several constants peculiar to web files, our prototype provides strong evidence that a faster solution exists.

Most research in the web caching field ties closely with that of networking. Huge strides have been made in the thickness and latency of network pipes. Other research has helped develop effective cache hierarchies [2]. In the near future, the time to pull cached data off of disk will become a large bottleneck in its overall retrieval. Our research in a fast file system for web objects seeks to minimize this bottleneck.

Our central tenants are rapid lookup and rapid retrieval: the upper levels of the cache need to know quickly if a file is cached and need to be able to inject its data into the network as rapidly as it can come off the platters. All file descriptors live in memory and do not need to be backed up -- a crash and recovery just entails more misses, not lost data. All of a file's data can reside contiguously and when its data does change it is merely deleted and recreated.

Figure 1 below portrays that in the sphere where the Caching File System (CFS) competes with UNIX (ext2 on Linux), it is doing well:



This shows ext2 normalized to 1 and CFS's relative time on the same number of contiguous creates, reads, and deletes of 1000 byte files. These results will hopefully encourage the reader to press on through the following sections: related work, implementation, experiments, future work, and conclusions.

2 Related Work

Data General's Nova computers supported contiguous file operations as did the original IBM VM/CMS operating system. While contiguous disk operations are very fast – almost zero seek time for a single read or write – there are several major problems with this type of file system. These include external fragmentation of the disk and determining how many blocks to initially allocate to a file [3].

These are two very significant problems with contiguous storage and explain why so little research has been done in this area. However, some substantial work on such a system was done by the researchers who developed the Amoeba distributed operating system [4]. Their system included files that were not allowed to change in size once they were written. This eliminates the allocation problem mentioned earlier. Application programs were only allowed to create, read, and delete files. Writing to an existing file was not allowed. To also enhance performance, this system stored administrative tables in memory. The result was a very high-performance file server called Bullet [5].

While this is very similar to CFS, the designers of Amoeba still had to be concerned with access rights to files and preserving file system data in the event of a crash. These issues were addressed by using a Directory Service. This system stored the capabilities for files on disk. Every file operation required that this capability be obtained first, then use this to index into the in-memory administrative table, and then access the files data from disk (or create the file, or delete it). This requires one extra disk access per file operation and interaction between two different entities (Bullet and the Directory Service). CFS, however, does not need to worry about these reliability issues or about getting a user's access rights. All files in a web caching system have a backup at the original file server. Also, only one program will ever be allowed to access the files on disk, so having to retrieve a files permissions list is not necessary. CFS is also a simpler system because it is a single entity.

3 Current State of Caching Web Objects

Squid is a commonly used and highly regarded web file cache. Most of its focus has gone into making its networking fast. Squid uses the operating system's default file system to store files (it runs on many platforms). Squid can sit on either end of the Internet: at an Internet Service Provider or corporation gateway or just in front of a web server.

Like our implementation, Squid keeps its records of files in memory. If CFS were incorporated into Squid, our additional data structures, like start block and byte length would merge into this structure. According to Squid's online documentation:

> Every object saved in the cache is allocated a StoreEntry structure…. Squid can quickly locate cached objects because it keeps (in memory) a hash table of all StoreEntry's. The keys for the hash table are MD5 checksums of the objects URI. In addition there is also a doubly-linked list of StoreEntry's used for the LRU replacement algorithm. When an entry is accessed, it is moved to the head of the LRU list. When Squid needs to replace cached objects, it takes objects from the tail of the LRU list…. Objects are saved to disk in a two-level directory structure [6].

In Squid's source code, it clearly does make the syscalls that we are testing our system against (in its disk.c in particular). As conveyed in the experimentation section, ext2 is really slow at deleting files. It takes almost five times as long to delete the files as read them in our tests. To help remedy this, Squid has an unlink daemon. Unfortunately, the daemon does not work when the system is very busy; to keep file usage consistent, it has to resort to calling unlink directly "when the cache swap size is over the high water mark." [7]

4 Implementation

4.1 Motivation

All files in a web cache share several characteristics. First, they never change size once they are written. This is probably CFS's most crucial invariant. It is intended to only work with files that can never be modified after the first file write operation. In fact, this system only provides three operations to perform on files. These are create, read, and delete. Each of these calls is discussed in more detail in section 4.4. In addition to being immutable, all files never change permissions and there exists a backup copy for every file at the original web server that provided the file. Of these last two points, the former means less metadata needs to be stored about the file. Given these invariants, the CFS file system is able to provide substantially superior performance to the Unix file system currently used on many caching web proxies.

4.2 Object Design

CFS was designed with the intention of allowing a higher level process to run multiple threads and have all of them asynchronously access the file system. This is accomplished by queuing up all requests that require either reading or writing to disk. A separate disk thread, provided by CFS, then processes each of these requests. The higher level thread that submitted the request is notified once the disk thread has completed the request. Using this mechanism, the higher level threads can submit a variety of requests without having to wait for its previous request to finish. A sync() call is provided so a thread can assure that its requests have been completed before it moves on. This mechanism is discussed in sections 4.4.5. Figure 1 below shows the basic layout of this system.

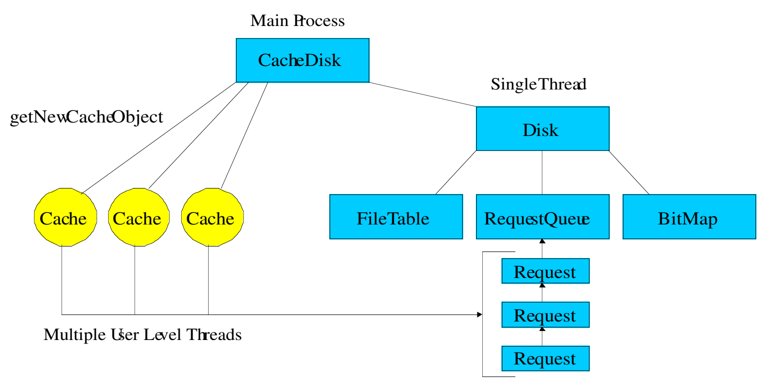


Figure 1 - Object Diagram

4.3 Directory Structure

CFS utilizes an in-memory data structure to keep track of all necessary information about a file on disk. The size of a file, its starting location on disk, an integer key for faster searches of the hash table, and the MD5 hash of the file's name, are all stored in a file descriptor element. All name lookups require only memory accesses. There is no reference to disk to find out information about where the file's actual data resides on disk. This saves at least one lookup over Unix. Unix must first check a file's inode information.

One main drawback to these in-memory file descriptors is that all the data about file locations will be lost in the event of a system crash. However, unlike a traditional file system that has the original and possibly only version of the file stored on its disk, all of files cached in a web proxy have their originals stored at a server somewhere on the Internet. All of these files can then be retrieved as requests are made to the web proxy. There would be a period of time where the browser would see a small performance degradation as the cache is reloaded. The data given to the browser, however, would be completely up to data and accurate. To allow planned server maintenance or shutdown, it would be a trivial matter to write this in-memory data to disk at an administrator's request. Then the system could be shutdown and re-booted without having to reload the cache.

4.4 User Level Requests

CFS provides a total of five different calls for a higher level application running as a proxy server. The following is a list of the five operations and a brief description of each.

```
int create(char* url, char* buffer, int size);
int read(char* url, char* buffer);
int remove(char* url);
int length(char* url);
int sync();
```

4.4.1 create

Create takes in the URL of a new file, the buffer in memory that stores all the data for the file, and the number of bytes in the buffer. The URL is converted into a different string using an MD5 hash. By requiring that all data to be placed in the file be first placed in memory, the largest file this system can store is limited by the size of virtual memory.

The first action of create is to find a location on disk to put the file. The disk thread provides a function to find an appropriate location on disk to store the file. It also provides another function to mark the necessary locations in a bit map as being full. Currently, CFS does not support writing data from multiple files to the same block. A single block on disk is completely allocated to a single file.

Create does not actually wait for the data to be written to disk. Instead, a request is generated and placed into a queue for the disk. The *create* function then returns and the thread can continue on doing other operations. When the disk thread actually services this request (actually writes the data out to disk), it signals the thread that generated the request if it is sync'ing. See Figure 2.

4.4.2 read

The first action *read* performs is to search the directory hash table for the appropriate file descriptor. Finding this, it then has the file's starting block on disk and the total length of the file. As with *create*, *read* does not wait for the data to actually be read from disk. Instead, it generates a request and places this into a queue for the disk. It then returns and the calling thread continues on. When the disk services the request, the thread that generated the request is notified.

4.4.3 remove

The *remove* operation is very simple – and very fast. The only thing that needs to be done to delete a file from disk is to remove the proper file descriptor from the directory and to modify the disk bit map to show the necessary blocks as now being free. Since all of this information is contained only in memory, these operations are very fast. Hence, unlike *create* and *read*, *remove* returns only after it has completed its operation. As the testing results in sections 5 will show, the *remove* operation is orders of magnitude faster than an unlink operation in the ext2 file system. This makes sense since ext2 must go to disk to delete a file.

4.4.4 length

This is a simple function provided to help the higher level threads in determining how much memory to allocate for a buffer to be used in a read call. It simply returns the length of an existing file (or an error if it does not exist). As with *remove*, this is a very quick operation and returns only after the necessary operations are performed.

4.4.5 sync

The purpose of this function is to allow a thread that has been submitting read and write requests to guarantee that all of its requests have been finished. The disk thread keeps track of how many outstanding requests each cache thread has. When this number goes to zero, the *sync* call returns and the thread is allowed to continue on.

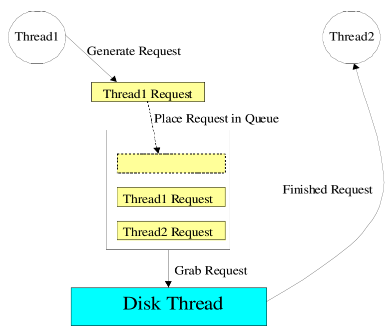


Figure 2 – create and read files

4.5 Physical Disk Interaction

Unlike other file systems that allow a file's data to be written to different parts of the disk, CFS requires that the entire file be placed on contiguous blocks on disk. Reading and writing from contiguous blocks utilizes the highest percentage of the disks bandwidth. The Log-Structured File System is an example of another file system that utilizes this feature [8].

Originally, CFS was to utilize a raw I/O patch from SGI to write directly to disk and completely bypass the Linux kernel. The first design of the system, however, used the Linux kernel buffers for all of its reads and writes. When the raw I/O patch was implemented, CFS sees a performance drop of about 6 to 7 times. The reason for this is that the system was not buffering any reads and writes in memory. This proves the obvious – buffering works. Given the time constraints in developing our system, we were not able to implement buffering. Hence, all of our results in section 5 are shown using the Linux kernel buffers for reads and writes.

5 Experimental Setup and Results

5.1 Experimental Environment

The Cache File System was implemented on a 500 MHz Pentium III dual processor with 1 GB of RAM, and 8.5 GB of disk space. Our initial tests and those presented in this report were run on a Linux 2.2.12 kernel using the ext2 file system. CFS uses the disk as a buffered block device (e.g. /dev/sde).

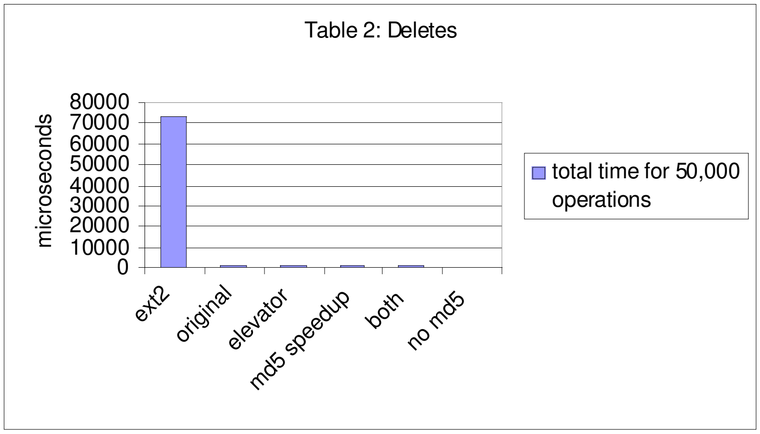
We ran a total of four benchmarks on each file system: three micro-benchmarks and one workload. Each microbenchmark (creates, reads, and deletes) consists of 50,000 operations with 1K files filled with random data. We produce on data point per 1000 operations.

The last test uses a trace file of actual browser requests [9]. It tracks all of the HTTP requests made to a proxy server over the course of one day: over 18,000 requests. Each of these requests is marked as a cache hit, a cache miss, or stale data that needed to be replaced. Our benchmark goes through this trace file one entry at a time and creates the file if it is not already on the disk or reads it from disk if it is there. If the trace entry indicates that the data to read was stale, we delete the file just read and create a new file. Our tests simply generate random data for the file instead of actually going out to a server and getting the real data. The data for all of these tests are graphed on the following pages. Section 5.2 discusses these results.

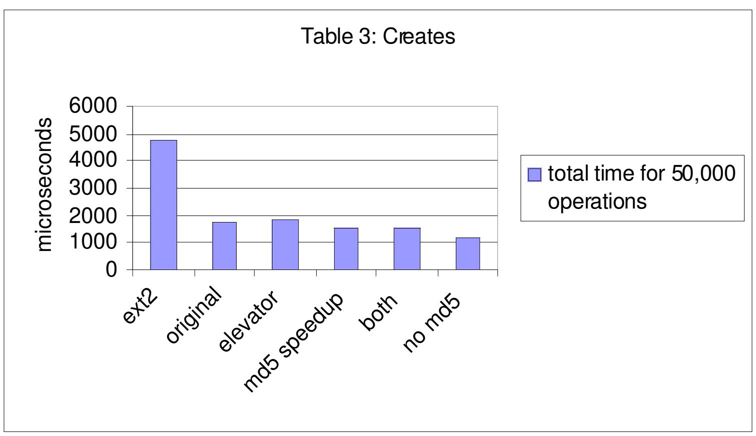
5.2 CFS verses ext2

The end result of all tests between the cache file system and ext2 was that CFS is faster in every test – even reads with our speedups. This should be no surprise considering that for every file that is created, read, or deleted, the CFS system makes at most only one access to disk. The Linux file system, ext2, however, will generally make two accesses to disk and maybe more. Of course, caching inodes and file data in memory will help Linux perform better, but caching files will also help CFS. Considering that the files being accessed are based on browser requests from users and are fairly random, the benefits of caching files in memory are going to be substantially reduced.

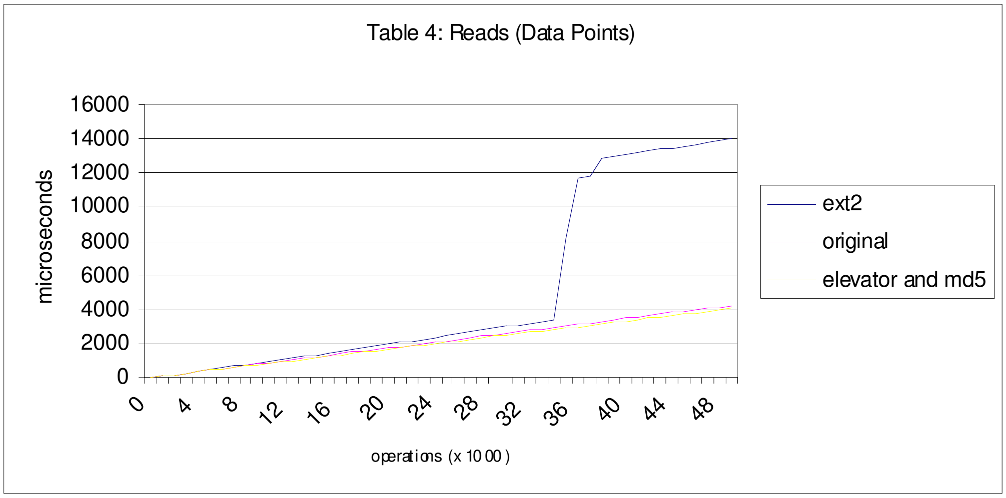
Of all the system calls made, delete has by far the greatest performance increase over ext2. Table 2 shows an increase of over 95% for CFS over ext2. This is due to the fact that a delete in CFS requires *zero* accesses to disk. Ext2 requires a minimum of at least one disk access. This is to remove the file entry from the directory inode that references it and make sure this update is stored on disk:



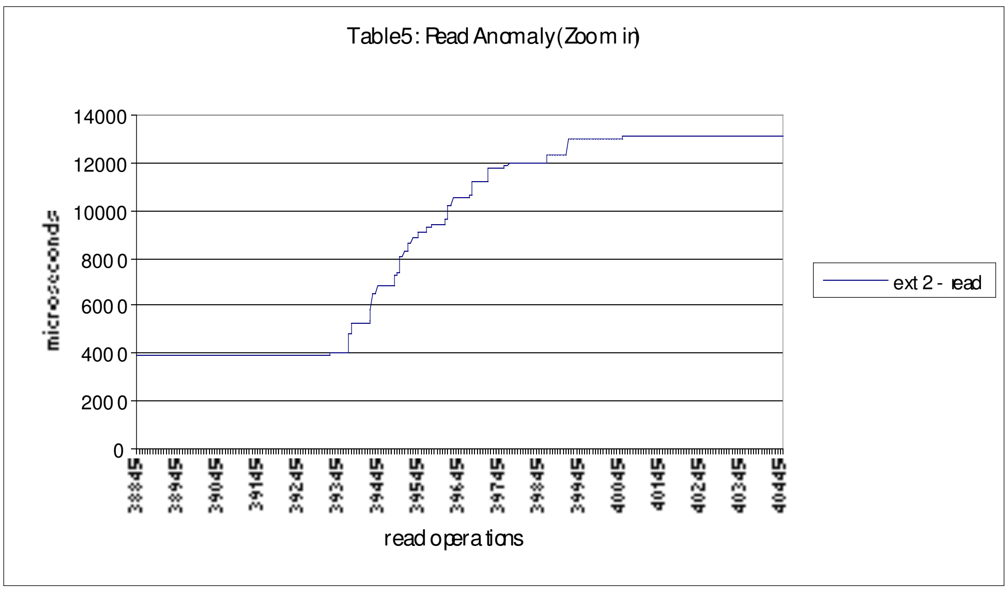
Creates also show a significant performance increase. Again, this is primarily because CFS has no structures on disk to update when a file is created. It simply writes the files data to disk and update the in-memory data structure describing the file. As with delete, ext2 must update an inode indicating where the start of the file is located on disk and where all of the other blocks are located. It must also add an entry to the directory inode that contains the file. This means two more accesses to disk for creating a file. Because of inode caching in ext2 and because writes in CFS require an access to disk, we do not see nearly the speedup that was seen in delete, but CFS is still over 60% faster than ext2 in this micro-benchmark. Table 3 shows this performance increase:



The read tests were a bit surprising. They show that for long stretches of time, ext2 performs equally as fast as CFS. This is almost to be expected because Unix like file systems have been highly tuned for sequential reads and they are very good at this type of file access [10]. However, for brief instances, the ext2 file system performs much worse than CFS. This short burst of poor performance appeared in every test we ran. One theory is that at certain times the buffers holding the data to be written to disk fill up and the file system temporarily has a significant slow down as these buffers are written out to disk. After this time, the file system again performs for a long stretch at a pace very similar to CFS. Table 4 shows one of these ext2 anomalies occurring. It also portrays that CFS's reads did get faster once we added in two of our optimizations:



A close-up of one of these ext2 hiccups occurring reveals that it is not one read which chokes, but that all of the read system calls slow down for a brief period:



A second hypothesis is that the cached directory inodes storing information on where each of these files are located on disk are all "used up" and a new set of directory inodes need to be brought in from disk. This extra disk access would also explain the huge jump in read time at set intervals. Again, due to time constraints we were unable to verify either of these theories or explore other possibilities. Table 5 shows this anomaly. If this anomaly is considered, CFS runs about 65% faster than ext2. If this anomaly is taken out, CFS runs marginally faster.

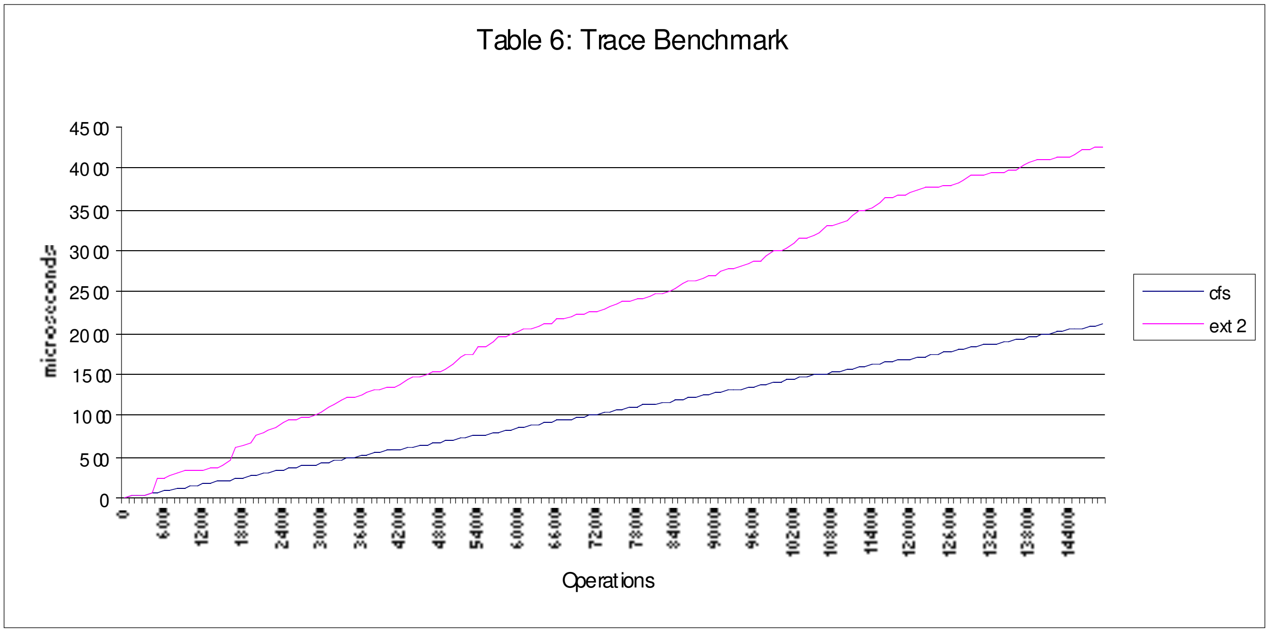


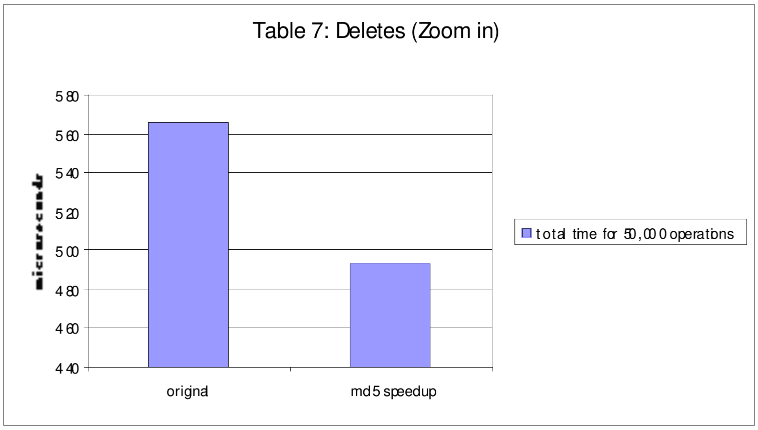
Table 6: Trace Benchmark

The final benchmark was the trace file. This was intended to simulate a real world workload for a proxy server. This benchmark randomly tests creates, reads, and deletes of each file system. In this benchmark, CFS runs approximately 50% faster than ext2. This is for all the reasons described above for each of the micro-benchmarks. Table 6 illustrates these performance differences.

5.3 CFS Tweaks

After our initial tests revealed that CFS was performing well, we decided to isolate the bottlenecks in the system. We also wanted to try character I/O, as this was one of the initial goals of the project. Using gprof, we found three primary bottlenecks: searching the bitmap for free space, probing the file table, and hashing URLs to MD5 digests. Additionally we believed we could reduce seek times and speed up read performance with a one-way elevator algorithm, although this surprisingly showed no performance benefit. Table 3 (create microbenchmark) even shows that sometimes the elevator algorithm's queuing overhead can slow operation as compared to the original system's FIFO. Speeding up bitmap access was left unresolved.

To improve searching the file table (a hash of file descriptors), we added an integer key to the file descriptor. This key is just a small chunk of the MD5 string. Traversing each chain of the hash table now involves an integer comparison instead of a *memcmp*. If the integer key of the file matches the integer key being requested, then a *memcmp* is performed to verify that the right element was indeed found. In other words, the integer key acts as a hint and the *memcmp* operation is then used to do an absolute verification. If we zoom in on our original delete microbenchmark (Table 2), the reader can verify that this hint makes lookups faster:



Because gprof revealed that about 15% of our CPU times were spent doing MD5 transformations, we decided to pull this out and do a simpler hash. The speedup is visible in tables 2 and 3. Unfortunately taking the MD5 hash out results in more skew in the file table and more complicated file descriptors, because they now need to hold strings of variable size.

5.3.1 Raw I/O: Buffers are speedy

Unbuffered character I/O proved surprisingly to be an anchor. Our microbenchmarks were repeatedly writing to disk instead of appending to an in memory buffer. To see how much slower these repeated writes were as opposed to one long one, we wrote a program to write out 512000 bytes as one large file and as 1000 512 byte files. The small files took 6 seconds as compared to the large file's 36 microseconds. An improved CFS would buffer data and use raw I/O.

6 Future Work

The most substantial part of research not included in this paper is how to handle the possible fragmentation of the disk over time. This fragmentation will arise as files in the cache are deleted. This is not a trivial problem to solve but there are a few properties of web files that may lead to fairly simple solutions. First of all, one reason for deleting a file is because the copy on disk has expired and needs to be re-retrieved from a server. There is a good possibility that this new version of the file is going to be very similar in size to that of the version just deleted. If this is the case, the new file can go directly into the spot on disk just vacated by its predecessor.

The most common method of dealing with the external fragmentation problem is to do a periodic compaction on the disk. The current version of CFS implements a very simple method of doing this. Every time a file is read, the bit map describing the disk is examined. If the area around the file just read contains a certain percentage of free blocks, the file is moved as close to the front of the disk as possible. This will obviously provide some performance degradation as the entire file must be read off of disk and rewritten to disk. This transfer could be done lazily to make the decrease in performance seem as low as possible to the browser. Another option is to simply delete the file. This would mean the next time the file was requested, it would have to be fetched from the server instead of the cache. The advantage to this method is that the new version of the file retrieved from the server would not only be more current, but it could be put in a location on disk that would make things more compact.

It would also be beneficial to provide an interface for a higher level process to find out which files on disk are old and ripe for replacement. This would be needed when the disk becomes full and a new object needs to be placed in the cache.

Research into using CFS in an actual proxy server implementation would be of great benefit in analyzing its performance. It would also be useful to run the above tests with more than one thread generating requests to the disk. Lastly, research into using a raw I/O patch with buffering could produce a system with even better performance.

7 Conclusions

The most important conclusion from all of this research is that specialization works when it comes to caching web pages. By taking advantage of immutable files, constant permissions, and back-ups of all files at an original server, the cache file system presented here is able to achieve speeds at least twice that of a Unix style file system doing the same operations. This cache file system is not nearly as flexible as other types of file systems, but it does not need to be. Its sole intent is to provide high speed operations on cached web pages. While work still remains to be done in dealing with fragmentation and providing an interface to higher-level programs so they can determine replacement policies, the core work presented here achieves the goals it set out to accomplish – creating, reading, and deleting files – substantially faster than Unix can.

References

[1] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang, *Optimistic Incremental Specialization: Streamlining a Commercial Operating System.* Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95), December 3-6, 1995, Copper Mountain, Colorado.

[2] L. Fan, P. Cao, J. Almeida, A. Broder*, Summary Cache: A Scalable Wide- Area Web Cache Sharing Protocol*, ACM SIGCOMM '98, pp. 254-265.

[3] Abraham Silberschatz, Peter Galvin, *Operating System Concepts, fourth edition*. Addison-Wesley Publishing Company, January 1995.

[4] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba, *A Distributed Operating System for the 1990s*. Computer, Vol. 23, No. 5, May 1990.

[5] Robbert van Renesse, Andrew S. Tanenbaum, and Annita Wilschut*, The Design of a High-Performance File Server*, IR-178, Vrije Universiteit Amsterdam, November 1988.

[6] http://www.squid-cache.org/Doc/Prog-Guide/

[7] http://www.squid-cache.org/Doc/FAQ

[8] Rosenblum, M. and Ousterhout, J., *The Design and Implementation of a Log-Structured File System*. ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, pp. 26-52.

[9]http://www.ircache.net/Cache/ - sanitized access logs in section on Statistics Data and Visualization

[10] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S., *A Fast File System for UNIX*. ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 181-197.