UCFS – A Novel User-Space, High Performance, Customized File System for Web Proxy Servers *

Jun Wang, Rui Min, Yingwu Zhu and Yiming Hu
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221-0030
e-mail: {wangjun, rmin, zhuy, yhu}@ececs.uc.edu
corresponding author: Yiming Hu
Phone: (513)556-4760
Fax: (513)556-7326

Abstract

Web proxy caching servers play a key role in today's Web infrastructure. Previous studies have shown that disk I/O is one of the major performance bottlenecks of proxy servers. Most conventional file systems do not work well for proxy server workloads and have high overheads. This paper presents a novel, **U**ser-space, **C**ustomized **F**ile **S**ystem called *UCFS* that can drastically improve I/O performance of proxy servers. UCFS is a user-level software component of a proxy server which manages data on a raw disk or disk partition. Since the entire system runs in the user space, it is easy and inexpensive to implement. It also has good portability and maintainability.

UCFS uses efficient in-memory meta-data tables to eliminate almost all I/O overhead of meta-data searches and updates. It also includes a novel file system called *Cluster-structured File System (CFS)*. Similar to the Log-structured File Systems (LFS), CFS uses large disk transfers to significantly improve disk write performance. However, CFS can also markedly improve file read operations, and it *does not generate garbage*.

Comprehensive simulation experiments using five representative real-world traces show that UCFS can significantly improve proxy server performance. For example, UCFS achieves 8–19 times better I/O performance than the state-of-the-art SQUID server running on a Unix Fast File System (FFS), 4-7.5 times better than SQUID on asynchronous FFS and 3–9 times better than the improved SQUIDML.

Index terms:
File systems. Web Proxy Servers. Disk I/Os. Performance Improvement.

* A portion of this work has been reported in a paper presented in the workshop of Performance and Architecture of Web Servers (PAWS2001) [1].

1 Introduction

The World Wide Web is a large distributed information system providing access to shared data objects. Web proxy caching is well-recognized as an effective scheme to alleviate service bottleneck, reduce network traffic and thereby minimize the user retrieval latency. Different kinds of proxy caches, such as cooperative hierarchical web caching architectures, large independent root caching servers and institution leaf-level caching servers are widely used.

Various studies have shown that Disk I/O is one of the biggest performance hurdles for large web proxy servers[2, 3]. The web cache hit rate grows in a logarithmic fashion with the amount of traffic and the size of the client population[4, 5]. Modern proxy servers have a very significant document miss rate (typically from 40% to 70%[6]). This means that 30% to 60% of incoming requests would generate disk operations to create new web documents on the disk. Moreover, proxy servers have a very low document hit rate in the RAM cache. Even with a large RAM, say 512MB, only 5% of all hits are in RAM [7]. Seventy to eighty percent of hits go to disk files (disk hits), resulting in a high demand on disk I/Os. Other hits are If-Modified-Since hits, namely IMS hits with the 304 reply code, which occur when a client requests a file with modification date later than a certain time. Finally, proxy servers have to invalidate stale files for maintaining consistency, generating frequent updates on the disk.

Most current proxy servers are designed to run on top of a general purpose file system like the Unix File System (UFS) or FFS. While these systems have good performance and many other advantages for general-purpose computing environments, using them with highly-specialized workload such as proxy servers will result in poor performance, as discussed below. We need better solutions to cope with the increasing demand on proxy server performance.

1.1 State-of-the-art Proxy Servers

1.1.1 SQUID and SQUIDML

SQUID [8] is a public web proxy cache software which is one of the most widely-used proxies in both research and real-world environments. SQUID uses the file system of the host operating system (such as UFS) to handle cached web documents on disks. An improved SQUID scheme using memory-mapped (*mmap*) files called SQUIDML can reduce disk I/Os by 50% [3].

1.1.2 Problems with Current Approaches for Proxy Servers

SQUID, SQUIDML and most of other proxy servers use regular file systems (such as FFS) and/or the *mmap* function to manage cached web documents on disks. However, such approaches have the following major drawbacks :

1. Studies have shown that 75% or more web documents in web proxy traffic are less than 8KB [3]. It is well-known that FFS and other conventional file systems cannot efficiently handle small files, leading to very poor performance [9, 10, 11, 12, 13, 14, 15]. For example, Rosenblum and Ousterhout [10] and Lumb *et al.* [16] pointed out that these file systems can only utilize 2–15% of disk bandwidth. Much of the disk bandwidth is wasted on disk seeking and rotational latencies because small disk I/O requests dominated in such systems.

2. Many file systems such as UFS uses synchronous writes for file meta-data updates in order to ensure file system consistency [1]. Furthermore, most systems periodically flush dirty data from the buffer

[1] In Linux, file systems can have an option of using asynchronous updates.

cache to the disk to improve data reliability. Both policies cause considerable performance degradation. Although they are critical for general-purpose systems, as we explain earlier, for most proxy caches, such operations are unnecessary, since data reliability is not a major concern. If data is lost, it can be fetched again from the original web server. There are some research file systems like Soft-Updates for FFS [14, 15] , journalling file systems [13] developed to ameliorate this problem but still with other limitations (to be discussed in section 1.1.3).

3. For FFS, a file read operation may suffer a high I/O penalty, especially in case of a directory lookup cache (DLC) miss. A file read may generate two disk reads for two-level directory inodes, one read for file inode, one write to the file inode (to update the file access time), and one or more reads for the file data itself. Since the proxy servers may consist of millions of files, the DLC hit rate is limited (less than 70% in our experiments). The DLC has also to compete with other meta-data and data caches for the limited memory size. For large files, there may be other I/O penalty of accessing indirect blocks.

4. SQUIDML uses *mmap* files to manage cached files on the disk. While *mmap* reduces some I/O overheads associated with the file system, further performance improvement is limited. As discussed before, most cached files (which are normally small) cannot fit into the RAM and have to be stored on the disk. Small and expensive I/O requests are frequently needed in order to swap these files in/out.

5. Another problem with SQUID is that SQUID hashes the URLs (Universal Resource Locators) into two-level directory entries. URLs pointed to the same web-site are likely to end up in different directories, destroying the host locality of the HTTP reference stream [3, 17]. Maltzahn *et al.* suggested using virtual memory for files smaller than the page size of virtual memory and preserving host locality to reduce disk I/Os [3]. However such a method has only limited benefits, as the virtual memory is not optimized to handle the proxy traffic. Moreover, for machines with 32-bit process address space, normally only 2GB is available to users. This method could only support 256,000 8K-sized web documents at most, unless a complex, multi-processing architecture is used.

1.1.3 Possible Solutions

In order to solve the above-mentioned problems, we can use the following possible solutions. However, each of them has drawbacks.

Log-structured File System(LFS) [9, 10, 11] can be used to significantly improve the performance of small writes. However, there are three major problems with this solution. First, LFS is not currently supported by any major OS. Second, LFS may suffer from serious performance degradation because of the garbage collection overhead when the workload is high. Seltzer *et al.* [18] pointed out that cleaner overhead reduces LFS performance by more than 33% when the disk is 50% full. The LFS performance will be degraded more quickly when the disk is near full. Finally, LFS does not improve read performance, which is also important for proxy servers.

Similar to LFS, Disk Caching Disk (DCD) [19, 12] uses large logs to improve small write performance. DCD works in the device or device-driver level, so it has better portability. However, DCD suffers from the destage overhead. This destage operation is a background process that transfers data from the cache-disk to the normal data disk. It also does not improve read performance.

Soft-Updates can improve file system performance by eliminating most overheads associated with synchronous meta-data updates. While it is a major improvement for general-propose file systems, Soft-Updates is currently supported only by a few operating systems (such as Free-BSD). More importantly, there are two of the fundamental problems with proxy servers it still cannot solve. One is the dominating small disk writes deriving from creating new small files. The other is that much of the regular file data (not the meta-data) cannot fit in the RAM and has to be accessed from the disk. Soft-Updates cannot solve these problems.

Moreover, Soft-Updates do not speed up reads. Journalling file systems and Metadata-logging [20] have similar problems. Finally, FFS could also work in the asynchronous mode (FFS-async) by using asynchronous writes for updating metadata. This improves performance at the cost of sacrificing consistency. But FFS-async can not solve the problem of small disk writes and does not improve disk read performance.

We may also develop a new file system in the kernel that is optimized for proxy servers to replace FFS. However, such an approach is very costly since it is difficult to implement and maintain a new file system in the kernel. It also poses a huge portability problem, as file system designs are very closely tied to kernel structures, which vary among different operating systems or even different versions of the same OS.

1.2 Our Solution

We notice that database systems face a similar problem. Since most file systems cannot satisfy the performance demand of high performance database systems, most database systems choose to manage their own data and file meta-data on raw disks.

In order to achieve significantly better performance, we propose to let the proxy server manage its own data and meta-data and by-pass the file system. Here the meta-data include those of proxy servers themselves and file systems. By using a dedicated raw disk (or a raw disk partition), we can design data and meta-data managing algorithms that are optimized for proxy server applications. Moreover, these algorithms run in the user-level as a part of a regular proxy server process. As a result, they are easy to implement and maintain, and have good portability. There is no need to change the kernel.

In the rest of the paper we describe the design of Web Proxy Server File System (UCFS), a user-space, customized file system for proxy caching and other network applications. UCFS is a user-level software component that can be used by a proxy server (or other server applications) to manage files. It uses efficient in-memory meta-data structures to eliminate almost all disk I/Os related to meta-data searches and updates. Furthermore, it employs a novel file system called *Cluster-structured File System (CFS)*, which takes the full advantage of the good locality of web accesses. CFS has a good write performance similar to LFS, as both use large disk transfers. Unlike LFS, CFS also significantly improves read performance. More importantly, CFS does not generate garbage and therefore has no cleaning overhead.

1.3 Organization of This Paper

The remainder of the paper is organized as follows. Section 2 and 3 describe the design and operation of UCFS. Section 4 describes our experimental methodology. Section 5 presents simulation results and analyses. Section 6 discusses related work. Section 7 summarizes the new strategy and Section 8 discusses future work.

2 The design of UCFS

2.1 Data-structure Definitions

In order to describe the design clearly, first we define several terminologies used in UCFS.

1. **RAM Buffer Cache**
   This is the user-level RAM buffer cache in UCFS, which is different from the I/O buffer cache of regular file systems.

2. **Disk-level Cache**
   This represents the disks used by proxy servers to store cached files.

3. **URL Request**
   This is the request received by the proxy server from Web clients/Web servers on HTTP protocols.

4. **URL File**
   This is the file identified by the host and path information in a URL request.

5. **File Meta-data**
   This consists of the file's attributes including the file name, size, last-access-time and etc.

6. **Cached File**
   Every URL web document will be cached in a unique file in proxy server. This file can be either in the RAM buffer cache or on the disk-level cache. It consists of the file meta-data and file data.

2.2 The System Architecture

The system consists of the following four major components, which are discussed in details in the rest of the section.

1. *An In-memory File Lookup Table*. This table contains the location information in memory or on disk for all cached files. It is used to track all files in the proxy cache.

2. *A Cluster-structured File System (CFS)*.

   This is the key of the whole design. A CFS works somehow similar to an LFS on writes: it tries to group many small writes into a fix-sized large write buffer and then write to the disk in a single large disk write, in order to achieve high performance by fully utilizing the disk bandwidth. However, there are major differences between CFS and LFS. In CFS, clusters are also the basic units for reads. Therefore all disk reads are large disk requests. This strategy makes CFS' reads much more efficient than small reads found in LFS (and FFS). More importantly, CFS does not generate any garbage on the disk, while garbage is the major problem of LFS.

3. *A Disk Cluster Table*. This table consists of the status of all disk clusters.

4. *A RAM Buffer Cache*. This cache holds hot files that will be accessed soon.

5. *A Locality-based Grouping Algorithm*. This algorithm identifies the *access locality*. Files which are likely to be accessed together (i.e., those from the same web site) are grouped together in one cluster, so they can be read from or written to the disk together. The rationale is that if one of the files is accessed, other files in the cluster are likely to be accessed soon.

Our proxy systems do not cache files larger than 256 KB. In the proxy logs we analyzed, these large files only take less than 1% of all files accessed. The servers also do not cache any dynamic documents because they are not cachable.

In the following subsections, we will describe each system component in details.

2.3 The In-Memory File Lookup Table

When a URL request comes in, the proxy server needs to quickly decide whether its URL file is cached, and if yes, where in RAM or on the disk is the file located.

Our design goal is that no disk I/O should be needed during this process. As a result, the file meta-data of all files in the proxy cache should be kept in tables in RAM. Since a large proxy server can cache millions of files, the tables must be very carefully designed to make sure that their sizes are reasonable.

The main challenge here is where to store the URL strings of the cached files. We could have maintained a hash table. Each cached file in the system has an entry, which contains the URL string, file location, present/absence information, and etc. URL string in the URL request can represent the unique identity for every URL file. When the server receives a URL request, we can compute the hash value of the URL, find the corresponding entry in the table, and compare the incoming URL request with the URL string in the hash table entry to decide if we have a hit or miss. However, such a solution may result in a hash table that requires a huge RAM space, because many URL strings are very long (up to hundreds of bytes). In all real-world proxy traces used in this paper, the average URL length varies from 65 bytes to 86 bytes.

2.3.1 The Solution: MD5

Our solution is to store only the MD5 checksums of URLs in the hash table, instead of saving the actual URL strings themselves. As shown in Figure 1, each entry of the meta-data hash table consists of 24 bytes plus one bit. An MD5 checksum of each URL string is calculated as the unique key [21]. This MD5 checksum occupies only 16 bytes in each hash table entry. Note that the actual URL string of each file is saved with the file data on the disk.

In addition, each entry uses another 4 bytes to store the disk cluster index, which is used to find the appropriate cluster (to be discussed shortly) on the disk that contains the file. A special "0000" cluster index shows that this file is not on the disk, only in memory.

We use an additional "MEM" bit to identify if this file has a copy in RAM. A "1" implies that this file has a memory copy. Note that it is possible that a file has copies in RAM *and* on the disk. This is for performance optimization and will be explained in section 2.4.4. Another 4 bytes save the "Last Modified Time" of this URL file, which is the value of the HTTP reply Last-Modified header. This information is used by the proxy server to determine if a cached file is stale or valid.

SQUID2.4 implements a similar hash table of all *StoreEntry's*. StoreEntry is an in-memory data structure to save file meta-data for cached files. Each StoreEntry consists of 56 bytes on "small" pointer architectures(Intel, Sparc, MIPS, and etc). In addition, there is a 16-byte cache key (MD5 checksum) associated with each StoreEntry. The StoreEntry could map to a cache directory and location via *sdirn* and *sfilen* two fields. Unfortunately, this policy only converts to a disk path name. To access the file, the file system still needs to perform expensive name translation in order to get the disk addresses.

MD5 Checksum (16Bytes)	Cluster Index(4B)	MEM(1Bit)	LastMod(4B)
0000 1111 2222 3333 4444 5555 6666 7777	1200	0	981666668
1234 5678 90ab cdef 1234 5678 90ab cdee	0000	1	981878888
1234 5678 90ab cdef 1234 5678 90ab cdef	1600	1	981888889
195c 77d1 ceff ed8e 2d2f e0fb 7634 6180	32688	0	986888888

Figure 1: The In-memory File Lookup Table

2.3.2 Performance and Space Requirements

With the in-memory file lookup table, all file meta-data searches and updates take place in RAM. File meta-data updates can be applied in memory because, as discussed before, data reliability is not a major concern for a proxy server. No disk directory searches or inode accesses are needed. This significantly reduces the overhead for both file writes and reads.

We use a new hash function for hash lookup table [22]. Comparisons usually take 7 instructions. Computing a hash value takes 35+6n instructions for an n-byte key. In our case n is 16 (the length of an MD5 checksum).

The in-memory file lookup table does take a large memory space. Currently we allow for up to 4 million cache entries in this lookup hash table. The total size of the table is less than 96 MB. However this is a small portion of the total RAM of a proxy server, which often has 512 MB of RAM or more. Moreover, as the price of RAM drops rapidly, systems will use even more RAM. Finally, our design is very compact compared to other systems. For example, a similar-sized SQUID hash table consumes around four times of RAM.

If this lookup table is full, which means there are 4 million files in proxy cache, proxy server will delete files from the proxy cache based on some replacement algorithm (See section 2.4.9 for details). When the files on disk-level cache have been deleted, their corresponding entries in in-memory file lookup table will also be reclaimed.

2.4 The Cluster-structured File System

CFS is the key to the entire design of UCFS. CFS divides the disk space into large, fix-sized *clusters*, which are the basic read and write units. The cluster size can be from 32 KB to 256 KB. Through simulation we find that a 64 KB cluster size has the best overall performance.

Note that CFS does not maintain any directory structures. All objects on the disk are files. This is because the directory information has already been taken care of by the in-memory file lookup table.

2.4.1 Disk Layout

Figure 2 shows the disk layout of CFS. Each disk cluster may contain multiple small files (which is the common case, as studies have shown that currently 75% of web documents are less than 8 KB). On the other hand, a large file can span multiple clusters.

The file meta-data consists of the following fields that are stored together with the file data:

1. URL string of the cached file, obtained from the HTTP request.
2. File Size.
3. Last-Modified-Time of the file.
4. Reference Number. This records the number of accesses of this file since being cached in the proxy.
5. Cluster Array. If a file is larger than a cluster, it will span over multiple clusters. This array records the cluster numbers of the remaining clusters of this file if the file size is larger than one cluster. The array has 8 entries. If the cluster size is 64 KB, then the maximum file size allowed is 9×64KB = 576 KB. This is more than enough, since our system does not cache files larger than 256 KB. The array contents are ignored if the file is smaller than one cluster.

In addition to the normal data clusters, each disk subsystem pre-allocates a super-block at the beginning of the disk to save important information such as the cluster size, the pointers to the disk copies of file lookup table and disk cluster table.

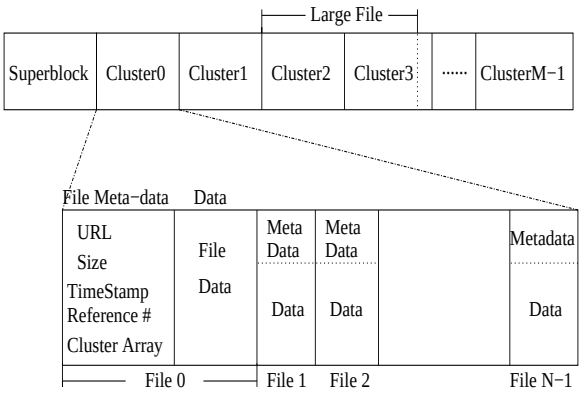

Figure 2: Disk-layout Table of UCFS

2.4.2 Writes

Writes in CFS are simple. When the write buffer is full, proxy server will evict inactive files from memory and put them into a write buffer in RAM. The size of the buffer is a multiple of the cluster size. When the write buffer fills up, the entire buffer is written to one or more disk clusters, in large, cluster-sized, disk writes. As in the case of LFS, such large writes in CFS are very efficient and have very good performance, because the seek and rotational overheads are relatively small compared to the data transfer time.

2.4.3 Reads

Reads in CFS are very different from those in other file systems such as LFS or FFS. Because clusters are the basic I/O unit in CFS, when reading a file, the entire cluster that contains that file is read, in a single large disk I/O, into the memory.

There are three important reasons of choosing such a design:

- First, reading an entire cluster that contains many files, instead of a single file, is in fact doing *prefetching* with very few extra overhead [2]. Since our system groups files into a cluster according to their locality (more on this in section 2.6), it is very likely that when one file in the cluster is accessed, some other files in the cluster will be accessed soon. By doing prefetching with a cluster read, we need only one disk access instead of many.

  Note that for most servers, both the *system throughput* (the number of processed requests per second) and the *system response time* are important. In CFS, prefetching can improve *both* disk response time and disk throughput, because it saves many small reads by loading many files in one single I/O. Performance will be improved even if only a few prefetched files are hit in RAM before they are evicted. In a regular file system such as FFS, prefetching can only improve the response time. It often has a *negative* impact on the overall disk throughput because of the increased number of disk requests. For example, if FFS prefetches 16 small files, it needs 16 or more *extra* I/Os. If only 6 of the 16 files are accessed later, 10 I/O requests are wasted and will hurt the performance.

- Second, we eliminate all file meta-data I/O overheads because we store them together with file data as an atomic storage unit on disk. The file meta-data has also been read into the memory at the same

[2] For example, for the disk we simulated, on average it takes *6.1ms* to read a 4KB file block. One the other hand, it takes only *7.1ms* to read a 64 KB cluster. This is because the disk seek and rotational latency, which are independent of request sizes, dominate the disk access time.

time when the file data is read into RAM buffer cache. Any future accesses, updates or invalidations to the file meta-data will be done in memory. For FFS, since the file meta-data are separated from file data and located on different disk locations, there are multiple I/O operations associated with the file meta-data when reading or writing files.

- Finally, cluster reads allow CFS to completely eliminate the problem of garbage, which is the major problem of LFS. Garbage is generated because individual files in a disk cluster (or segment in LFS) are invalidated. In CFS, after a cluster is read into the RAM, the entire disk cluster is marked as blank, because the files are now in RAM. If the proxy server has to invalidate a file on the disk because it is stale, the file (hence the cluster containing the file) is always read into the memory first, since the file meta-data of the file has to be checked anyway. Since files are now only in RAM and not on disk anymore, *invalidation to these files happen only in RAM, never on the disk.* As a result, garbage will never be generated.

2.4.4 An Improved Read/Write Policy

The above description on prefetching read is a little bit simplistic. In our actual design we do not immediately invalidate a disk cluster after a hit file is read into the RAM. Otherwise when all files in this cluster in RAM become the least recently used and has to be moved back to the disk, we have to write these files back to the disk, even though none of the files have been modified since last read.

After we read a cluster into the memory, we just mark the cluster as both in RAM and on disk. This operation is done in the disk cluster table. At the same time, we set the "MEM" field of in-memory lookup table as "1" for each file on this cluster. This shows all files on this cluster have another valid copy in memory from now on. The "cluster index" field of in-memory lookup table still records the previous cluster number where each file is located on disk. We can consider these files of this cluster a virtual cluster in RAM buffer cache. Files belonging to the cluster may or may not stay together in the RAM and will not be re-grouped, unless one of them is invalidated, updated or regrouped. Later on, if one file on the cluster has to be evicted back to the disk, CFS will first check whether the other files on this cluster are all in the cold buffer and intact since the last prefetching. If so, CFS simply marks the cluster as on-disk only, and deletes the files from the RAM buffer cache. For each file on this cluster, the "MEM" field of in-memory lookup table is set back to "0" to show that this file has only one valid copy on the disk. The unchanged "cluster index" field still records the correct cluster index for every file. No additional write back disk I/O is needed.

If, however, one of the files on this cluster is invalidated or updated, the disk cluster is invalidated immediately (by marking it in the disk cluster table in RAM). The "cluster index" of in-memory file lookup table will also be cleared to "0000". This shows that the prefetched files have only one valid copy in memory now.

This improved scheme saves one disk write-back operation by keeping two copies, disk and memory for each file. We have to implement additional tags and controls for disk cluster table and in-memory file lookup table.

2.4.5 Quantifying the Overhead of Large Prefetch Reads

When the proxy server finds a requested file on the disk, it will read the whole cluster containing the file into the RAM buffer cache. We have pointed out that such a large read is in fact doing prefetching with minimal extra cost. For example, if the cluster is 64 KB and has 16 files (4 KB each), then the time to read all 16 files in a single large read (7.1 ms) is close to that of reading one small file (6.1 ms). In other words, the system prefetches the additional 15 files into RAM with a very small overhead of 1 ms.

However, there is another hidden cost here. The additional 60KB of files prefetched will replace 60KB of files in the memory and generate an additional large write to write the replaced files back to the disk. If the

prefetched files are accessed in the future, the extra large write is not an overhead, since such replacements are necessary. Since not all files prefetched will be hit in the future, some overhead is unavoidable.

To evaluate if it is worth to perform large reads in CFS with such an overhead, we calculate the total I/O cost of a large prefetching read. We then compare the cost with that of the FFS operations needed to complete the same job. If the cost of CFS is smaller than that of FFS, it is worthwhile to implement such a policy.

A typical CFS cluster has a size of 64 KB and contains 16 small files (the average file size is 4 KB). In CFS, the *worst-case* I/O latency of reading a hit file from the disk-level cache is calculated by:

$$I/O\ Latency_{CFS} = 64\ KB\ disk\ read\ latency + 64\ KB\ disk\ write\ latency$$

The first factor on the right side of the equation is the overhead of the large read. The second factor is the overhead of writing the replaced files back to the disk. This result represents only the uncommon, worst-case overhead for CFS. For those prefetched files that will be used, replacing other files are necessary so the write-back process is not an overhead. Moreover, as discussed in Section 2.4.4, in CFS, this write operation can be avoided if none of the files in the cluster is modified.

Among the 15 files prefetched along with the hit file, several of them will be accessed by clients while these files are in memory. These files are defined as *Prefetch Hit Files*.

To calculate the prefetch hit rate, we add one prefetch tag in the file meta-data for each file to indicate if it is a prefetched file or not. We use a memory variable to count the total hits for all prefetched files in RAM. This number does not include the repeated hits to the same file. We then calculate the Prefetch Hit Rate:

$$Prefetch\ Hit\ Rate = \frac{Total\ Number\ of\ Prefetching\ Hits}{Total\ Numberof\ Prefetched\ Files}$$

From our experiments, we found that this hit rate varied from 28% to 38%. The mean is 33% with 5% standard deviation. These numbers are discussed in more details in Section 5.3.

Since the average Prefetch Hit Rate is 33%, and 15 files are prefetched on a large read, then on average, $33\% \times 15 = 5$ small files prefetched will be accessed by clients.

We now calculate the total I/O latency of FFS to complete the same amount of work: reading 6 small files (1 hit file and 5 prefetched files).

In SQUID/FFS, as we discussed before, each small file read from disk may result in several additional file meta-data I/O operations: two reads for two-level directory inodes, one read for file inode and one write to the file inode (to update the file access time). Although FFS implements a Directory Lookup Cache to capture repeated accesses to these inodes, the DLC hit rate is limited because a busy proxy server may accommodate millions of files. In our experiments, the DLC miss rate varied from 30% to 40%. We assume that the DLC miss rate is 35%. Each DLC miss I/O penalty for FFS file read is:

$$DLC\ Miss\ Penalty = 3 \times 4\ KB\ disk\ read\ latency + 4\ KB\ disk\ write\ latency$$

Therefore, the total I/O latency of FFS to read 6 small files is computed as follows:

$$I/O\ Latency_{FFS} = 6 \times 4\ KB\ disk\ read\ latency + 6 \times DLC\ miss\ Penalty \times 35\%$$

For the disk model we used, reading/writing 64 KB and 4 KB of data take 7.1 ms and 6.1 ms, respectively. We therefore can calculate the saving factor of disk latency for one typical large prefetch read:

$$I/O\ SpeedupTypical = \frac{I/O\ Latency_{FFS} - I/O\ Latency_{CFS}}{I/O\ Latency_{CFS}} = 519\%$$

This result indicates that large reads in CFS will have very good performance. Please note that this result is conservative, because we used the worst-case overhead of CFS here.

In the unlikely situation of Prefetch Hit Rate = 0%, the speedup of one large prefetch read over a small FFS read can be calculated as following:

$$I/O\ Latency_{FFS} = DLC\ miss\ penalty \times 35\% + 4\ KB\ disk\ read\ latency$$

$$I/O\ SpeedupWorst = \frac{I/O\ Latency_{FFS} - I/O\ Latency_{CFS}}{I/O\ Latency_{CFS}} = 3.1\%$$

So even in such unlikely situation, CFS will still have slightly better read performance than FFS.

2.4.6 Large Files

If a file is larger than the size of one cluster, it will span over several clusters. These clusters do not need to be contiguous on the disk. We use a "cluster array" in the file meta-data (which is located in the first cluster of the large file) to indicate the cluster numbers of the remaining clusters. The contents of the array is ignored if the file is smaller than a cluster. Large files are not common in proxy server workloads. For the traces we used, less than 1% of files are larger than 64 KB but smaller than 256 KB (files larger than 256 KB are non-cachable).

UCFS will always have good performance when reading and writing large files across several clusters. The system always uses large disk I/Os for these files. The minimal disk request size is the cluster size, and the request size will be larger if clusters are stored on the disk contiguously. FFS and other traditional file systems, on the other hand, may use small and expensive I/Os to handle these files if the disk is fragmented.

2.4.7 Other Data Structures of CFS

CFS uses an in-memory *Disk Cluster Table* to manage all clusters in the system. Each cluster in the system has an corresponding entry that includes the following fields:

- Last-access-time since the last CFS access or update
- Reference counter. This is the total reference number of the cluster.
- Valid status. This flag indicates if the cluster is "empty", " valid disk only" or "valid both on disk and in memory".

Since the in-memory file lookup table can track 4 million files at most, the maximum data capacity in CFS can reach 16 GB if the average file size is 4 KB. There are around 250 thousand clusters on the disk-level cache. Assuming each entry in the disk cluster table occupies 5.5 bytes, then the total memory requirement for this disk cluster table is less than 2 MB.

2.4.8 One-Timer Documents

As reported in [23], many files are accessed only once during their life time in the proxy server, so it is not worth caching. Previous research found that one-fourth of the total requests are for these one-timers and approximately 70% of the files referenced are one-timers.

The problem is that we cannot immediately know if a cached document is an one-timer or not. We can only wait for a specific time period (say several days). If we find a cached file has been accessed only once at the end of the period, we can assume that it is an one-timer file. Then we can delete it so as to free memory or disk spaces.

2.4.9 Replacement Policy

If after 115 hours [3], none of the files in the cluster are accessed, the cluster is invalidated. During simulation, we found around 30% of clusters consist of only one-timer files. The system also deletes old clusters from the system if the disk becomes full, using an LFU-aging algorithm [24]. When the UCFS replaces or deletes such files from the disk clusters, their meta-data entries in the in-memory file lookup table will also be updated or emptied.

CFS also maintains a variable in memory that records the *current disk head position* (a disk block address) called CDP. When CFS has to write some clusters to the disk, it always checks the Disk Cluster Table to see whether the CDP's neighbored space has enough free clusters to accommodate the incoming write requests from the RAM buffer cache. This strategy allows issuing of a very large disk write that consists of contiguous multiple clusters. It maximizes the use of disk bandwidth and saves some disk seeks and rotational latencies.

2.5 The RAM Buffer Cache

2.5.1 Design and Implementation

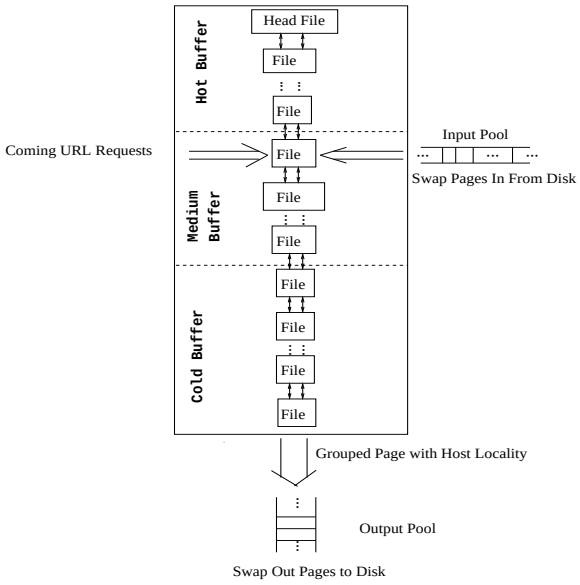

Figure 3: The RAM Buffer Cache of UCFS

The architecture of RAM buffer cache is shown in Figure 3. The objects in RAM buffer cache are only files. There are no "real" clusters in RAM buffer cache. As described in section 2.4.4, the system maintains a virtual cluster concept to save some write-back operations to the disk.

The RAM buffer cache holds the most active files in memory. In addition, when the RAM buffer cache is full, it works together with a locality-based grouping algorithm to group cached files into different clusters according to their host and temporal locality, before writing them to the disk.

All files in the memory are stored in the buffer in an LRU (Least-Recently-Used) list, with the Most-Recently-Used (MRU) file on the top. Note that we do not cache files larger than 256 KB in the proxy server.

[3]The number was chosen based on the result of SQUID cache age reported in [6]

We implement a hierarchical RAM buffer cache with three logical buffers according to the capacity thresholds: a *hot buffer* , a *medium buffer* and a *cold buffer*, which take 30%, 40% and 30% of total RAM buffer cache capacity, respectively. The buffer uses three head pointers and three end pointers to maintain the logical division on the doubly linked LRU list. Three memory variables are used to control the capacity of three logical buffers.

These three logical buffers work as follows:

1. Hot Buffer

   The most recently accessed files are put in the hot buffer. If the hot buffer is full, it will migrate the LRU file of hot buffer to the medium buffer.

2. Medium Buffer

   When CFS reads a cluster from the disk, it actually prefetches many small files in the same cluster to the memory. The cluster (hence the prefetched files) is first loaded here. If some of these prefetched files receive hits later on, they will move to the hot buffer. Also, when the proxy server sees a cache miss, it will create a new file and fetch the content of the file from the web server. The new file is also placed in the medium buffer. We put the newly created files and the prefetched files to the medium instead of the hot buffer because these files are potential one-timers, and we do not want them to pollute the hot buffer.

   When the medium buffer reaches the up-limit capacity, it will migrate the MRU file to the hot buffer and replace the LRU file to the cold buffer.

3. Cold Buffer

   Files in the cold buffer are the least recently accessed and are to be evicted out of memory. When the buffer is full, UCFS groups files based on their locality into several large clusters in this cold buffer. UCFS swaps these full clusters out of memory into the output pool. CFS fetches the ready clusters from this pool and writes them to the disk in one or more large disk writes.

By implementing such a three-level RAM buffer cache, we can successfully prevent the one-timer file from polluting hot buffer and keep the real “hot” files in RAM buffer cache.

2.6 The Locality-based Grouping Algorithms

One of the key assumptions during the design of CFS is that files residing in the same cluster have good locality. If one file is read, it is likely that other files in the same cluster will be read soon. This assumption allows us to adopt the cluster-sized large read policy that allows prefetching and eliminates garbage.

2.6.1 Reference Locality of HTTP References

In order to validate our assumption, we analyzed several HTTP log traces of proxy servers. We investigated each user’s access pattern in his/her every web session. We found that while a user may visit several different web sites at the same time, he/she would typically retrieve several related files from the same web site. This is because an HTML file often has many embedded files such as image files and audio/video files that will be accessed together. Moreover, the HTML file also contains hyper-text links to other HTML files, which are likely to be accessed in the future by the same user. It is such kind of *access locality* that we try to capture for grouping.

We conducted several experiments to validate this observation. We chose a HTML file accessed by the user, collected all the files accessed by this user within a short period (less than 10 minutes) into a cluster.

We then selected another user who visited the same web site. If this HTML file is accessed again by the new user, we found that he/she will access nearly half of all files in the cluster.

Previous studies have also shown the similar results we explored: web proxy accesses indeed have good access locality for each web client session [3, 25, 26].

2.6.2 Two-level URL Resolution Grouping Algorithm

We have developed a simple and efficient algorithm called *two-level URL resolution grouping algorithm* based on the above investigation. The details of this algorithm can be found in Figure 4 and Figure 5.

Algorithm 1

Algorithm to decide first evicted file and allocate the write buffer

Procedure AllocateWriteBuffer;

Begin

Create a file pointer fp;
Move fp to the LRU file in the *LRU list*;

While (the current file is within the half capacity of *Cold Buffer*)
AND (The extension of LRU file is not HTML)
Begin
Move fp to the next file in the *LRU list*;
End
EndWhile

If (the current file is not HTML)
/*There is not a HTML file within half capacity of *Cold Buffer**/
Then
Move fp to the LRU file in the *LRU list*;

Allocate Size of Int(filesize/clustersize)+1 clusters for *write buffer*;
Fill this file into *write buffer*;
Record this file as the first evicted file;
Release this file from *LRU list*;

End of AllocateWriteBuffer.

Figure 4: Allocate Write Buffer Algorithm

To start a group, the system first chooses a file that will be swapped out. Based on this first file to be evicted, we allocate a write buffer for this grouping. The size of the write buffer is one or multiple clusters. This process is described in Figure 4.

After the system decides the first file and allocates the write buffer, it would choose the candidate files based on their locality associated with the first evicted file within the scope of the half cold buffer, and put them into the write buffer. This process is described in Figure 5. If the write buffer is still not full, the algorithm selects files with same access frequency as that of the first evicted file or any small files (less than

Algorithm 2
Algorithm to group files based on locality into *write buffer*

Procedure Grouping
Input Parameter: the first evicted file
Output Parameter: write buffer

Begin
Create a file pointer fp;
Move fp to the LRU file in the LRU list;
/*First iteration, we fill files with the good locality*/
If (The *write buffer* space is NOT full) **Then**
While (The current file is within the half capacity of *Cold Buffer*)
If (the file has the same web site host/path) **AND**
(*Write Buffer* is large enough to save the file)
Then
Fill the file into *Write Buffer*;
Release the file from *LRU list*;
Else
Move fp to the next file from *LRU list* ;
EndofWhile
Endif
/*Next iteration, fill small files in *write buffer* to reduce the internal fragmentation*/
While (Free space of the *write buffer* is larger than 1 K Bytes)
Find files with same access times as that of first evicted file or small files with less than 1024 bytes size to fill in the *write buffer* within the half capacity of *Cold Buffer* using first-fit policy;
EndofWhile;

End of *Grouping*.

Figure 5: Locality-grouping Algorithm

1KB) until the free space of this write buffer is less than 1024 bytes. By filling up each cluster, the algorithm reduces the internal fragmentation within a cluster.

When selecting the first file to evict, the algorithm gives preference to HTML files. This is because that an HTML file is likely to have many related files to be accessed together, including embedded images and files pointed by embedded URLs in this HTML file.

3 UCFS System Operations

The previous section describes the design of each UCFS component. In this section we show how all these components work together.

3.1 Request Processing

When the proxy server receives a URL request from a client, the server passes the URL to the UCFS. UCFS then calculates its MD5 checksum and searches the in-memory file lookup table. There are several possible outcomes:

1. *A Cache Miss*. The file is not cached anywhere in the server. In this case, the proxy will fetch the file from its original server and put it in the medium buffer.

2. *A Memory Hit*. The file is cached in the RAM. The proxy server decides that the file is still valid. The file is transferred to the client directly.

3. *A Memory Stale Hit*. UCFS finds that the file is cached in the RAM. But the proxy server decides that the file is not valid anymore. The proxy server then replaces the file in RAM with the latest version fetched from the original web server.

4. *A Disk Hit*. UCFS finds that the file is cached on the disk. The proxy server decides that the file is valid. UCFS then finds the cluster containing the file, and loads the entire cluster into the medium buffer. The cluster on the disk is invalidated. The server then sends the file to the client. Since a cluster may contain many small files, these small files are prefetched into RAM.

5. *A Disk Stale Hit*. UCFS finds that the file is cached on the disk. But the proxy server decides that the file is not valid anymore. UCFS/CFS then fetches the cluster that contains the file from the disk to RAM and invalidates the disk cluster. Then, the server replaces the file in RAM with the new version fetched from the web server. Such a design does not generate any garbage, as invalidation happens only in RAM. Also, other files within the cluster are prefetched into the RAM. These files are likely to be accessed soon.

6. *An IMS Hit/Miss*. Web proxy servers have some part of IMS requests. These requests are typically used to verify the validate/expiration status for files among web clients, proxy servers and web servers. They may affect the networking performance but do not generate any I/O overheads. UCFS only processes them for checking the stale status of files.

3.2 Valid Hits and Stale Hits

In a proxy server, a cache hit (in memory or on the disk) can be either a valid hit or a stale hit. A valid hit means that the cached file is a valid copy of the original web object. A stale hit means that the cached file is out-dated and no longer valid. The proxy server has to re-fetch the file from the web server.

Each URL file in the system has a last-modified-time stored in the in-memory lookup table and a time-stamp associated with file data. The proxy can use these two values and the expire time from the server reply headers to communicate with the web server to decide if the file is valid or stale. The policy is similar to that algorithm adopted in SQUID.

3.3 Power Cycles and Re-boot

We have pointed out that for most proxy servers, data reliability is not a major concern. Even if all the cached files in a server are lost in a crash, they can be retrieved from their original web servers. This should not be a major performance degradation, because once a file is fetched, subsequence accesses to the file are satisfied in the proxy cache.

In fact, in a real proxy server product, crashes should be a very rare event. This is because that SQUID is most likely to be run on Unix systems which have been proven to be highly reliable. Power failure problems can easily be solved with an inexpensive UPS (Uninterruptable Power Supply).

UCFS can deal with regular shutdown/re-boot process though. If the server needs to be shut-down for maintenance or if the UPS detects a power failure, before UCFS shuts down, it saves the in-memory file lookup table, disk cluster table and all files in memory to the reserved superblock space on the disk.

After the proxy server restarts, UCFS simply reads the important data structures and files from the superblock space in the same order being written to the disk. These cached files are ready to be used and need not to be fetched again from the servers.

4 Experimental Methodology

We used HTTP-trace-driven simulation to evaluate the performance of our design compared to three baseline systems: *SQUID 2.4/FFS, SQUID 2.4/FFS-async* and *SQUIDML/FFS*.

SQUID 2.4/FFS is SQUID 2.4 running on top of a regular Unix FFS file system. SQUID 2.4 is the most popular, state-of-the-art web proxy server as of this writing. FFS is one of the most popular Unix file systems. *SQUID 2.4/FFS-async* is SQUID 2.4 running on top of a Unix FFS mounted in the asynchronous mode. Asynchronous FFS uses asynchronous writes to perform disk updates. It avoids all the overhead of synchronous updates of traditional FFS, therefore it has much better performance. However the file system consistency may be sacrificed in the event of a system crash. *SQUIDML/FFS* is SQUIDML running on FFS. SQUIDML has much better performance than SQUID because it uses *mmap* and virtual memory operations to reduce disk I/Os for small files. Together, these three baseline systems represent the state-of-the-art proxy server technology that are widely used.

4.1 The UCFS Simulator

We developed a UCFS system simulator on top of DiskSim [27], which is a widely-used, comprehensive disk simulator. We had to update the disk models slightly in order to simulate the state-of-the-art disk drives. We simulated a 27GB disk drive. We chose the Quantum Atlas 10K 9.1GB drive as the basic model (which is the largest model currently provided by DiskSim) and tripled the sectors of each track to expand the capacity. It has 478 megabit-per-second (Mb/sec) internal transfer rate [28].

In order to have an accurate comparison, we assume that the system has 512MB of RAM, which is a quite standard memory size for prevailing proxy servers. The trace logs we used in this paper were collected from five real-world large proxy servers with exactly the same 512 MB RAM configuration. See Section 4.3.1 for details.

Of this total 512MB RAM, 256MB is used to cache the file data. The remaining 256MB RAM is reserved for hash-table and other OS overhead (We are a little bit conservative here, because in reality the hash table of UCFS takes at most 96MB, leaving more memory for caching file data.). The CFS cluster size is 64KB unless otherwise stated. We chose this size because our results show that it has the best performance, as will become evident later. The proxy server simulator processes a comprehensive list of HTTP URL requests, including TCP_HIT, TCP_MEM_HIT, TCP_MISS, TCP_REFRESH_HIT, TCP_REFRESH_MISS, and etc. These requests are described in details in [29].

4.2 Simulators for Baseline Systems

While SQUID has implementations available, it is not possible to measure their performance with traces. As a result, we built simulators to use the same traces and disk models of UCFS so that we can compare their performance.

4.2.1 SQUID/FFS and SQUID/FFS-async

We assume that SQUID2.4 runs on a UNIX FFS file system. We also assume that the system has 512 MB of memory (same as UCFS). Since it is reported [8] that SQUID uses nearly 50% of RAM to store the

meta-data (the hashing table and URLs), we assume that 50% (256MB) of the RAM is used by SQUID to cache data, the remaining RAM is used for the meta-data and other OS overhead.

SQUID uses FFS to manage files. It translates each URL into a unique, numeric file name by hashing it into a two-level file path and saves it on the disk. A UNIX FFS simulator, ported from the BSD FFS design, was used on top of the DiskSim simulator. The FFS simulator provides detailed simulation to guarantee valid simulation and a fair comparison. It simulates FFS cylinder groups (each cylinder group has its own inode regions and free blocks management), FFS fragments management (4KB disk block sizes and 1KB fragment sizes), and a directory look up cache to improve the meta-data look-up. A hit rate of about 60-70% is observed during simulation.

For implementing FFS -async, we ignore the sync or fsync tags, which show synchronous writes in the I/O buffer cache. All writes are asynchronous in FFS -async mode.

4.2.2 SQUIDML/FFS

Maltzahn *et al.* presented a new way to reduce disk I/O by 50% compared to SQUID. Their idea is to circumvent the file system abstractions and store cached files into a large memory-mapped file. Disk space of the memory-mapped file is allocated once and accesses to the file are entirely managed by the virtual memory system. Specifically, for URL requests that are smaller than 8KB, they used *mmap* to process the requests. All other documents larger than 8KB are still handled by a modified SQUID cache architecture. It stores cached objects of the same server in the same directory assuming cache references to linked objects will tend to access the same directory. They use five segment sizes from 512 Bytes to 8K Bytes to retrieve objects from the memory-mapped file. A frequency-based cyclic replacement algorithm is used. They used trace-driven simulation to study the idea.

We developed a SQUIDML simulator to realize such a scheme. A virtual memory simulator and FFS simulator are built on top of DiskSim simulator. Files larger than 8KB are handled by the FFS simulator, while those smaller than 8KB are processed by the virtual memory through a *mmap* file.

In our simulation, we assume that the total system memory is 512 MB, the same as that in SQUID/FFS and UCFS. FFS I/O buffer caches and *mapping* share half of the memory (256 MB). The remaining RAM is reserved for in-memory hash table, meta-data, and other SQUID and OS overheads.

4.3 Workload Models

We chose several large suites of real-world web proxy traces to evaluate our system.

4.3.1 Large Web Proxy Server Traces

We used a set of traces from NLANR caches [29] which have been widely used in network research fields. NLANR traces consist of traces from seven NLANR caches. PA and SJ are self-service or independent proxy caches. The other five web proxy servers are organized in a hierarchical web proxy server architecture. They will communicate with other sibling servers for web documents. Their traces may show less locality than that of the two independent web proxy servers. The five caches receive 0.4-1.2 million requests per day, serving 5–14GB of data to 125–200 client caches throughout the world.

We obtained five of the seven traces from NLANR. The configurations of the five caches are summarized in Table 1. All five traces consist of logs for a two week period from 02/22/2001 to 03/07/2001. We were unable to retrieve complete traces of the remaining two caches during that period because of an unstable ftp server.

Name	City	RAM (MB)	Cache Size(GB)
BO1	Boulder	512	16
PA	Palo Alto	512	26
PB	Pittsburgh	512	16
SJ	San Jose	512	16
UC	Urbana-Champaign	512	16

Table 1: Configurations of the Five NLANR Caches

Name Name	Total URL Requests	Total Data Traffic(GB)	Document Hit Rate(%)
BO1	3,928,770	120.8	44.4
PA	4,990,283	115.2	57.7
PB	8,547,397	215.4	57.8
SJ	676,836	14.8	55.4
UC	8,107,179	253	47.8

Table 2: Characteristics of Five NLANR traces from 02/22/2001 to 03/07/2001

4.3.2 Preprocessing of Traces

NLANR access log traces consist of a comprehensive list of information, some of which are not cached by proxy servers at all. To speed up simulation, we simulated only those requests whose replies are cachable as specified in HTTP 1.1 [30]. We filtered out all CGI dynamic document requests, and other unrelated TCP requests like TCP_DENIED, TCP_CLIENT_REFRESH_MISS AND TCP_SWAPFAIL_MISS. We kept TCP_HIT, TCP_MEM_HIT, TCP_MISS, TCP_REFRESH_HIT, TCP_REFRESH_MISS, TCP_REF_FAIL_HIT, TCP_IMS_HIT and TCP_IMS_MISS HTTP requests.

Table 2 shows the statistic summary of the five post-processed NLANR web proxy server traces.

In order to measure the system performance in the stable state and avoid the transient effect, we used the first 150,000 requests to warm-up the system and started measuring performance after that point.

5 Simulation Results and Performance Analysis

In this section we compare the I/O performance of UCFS with that of SQUID/FFS, SQUID/FFS -async and SQUIDML/FFS.

5.1 Disk I/O Performance

5.1.1 Total Disk I/O Latency

Figure 6(a) compares the total disk I/O latency during the two-week period. Figure 6(b) shows the speedups of UCFS over the three baseline systems, calculated by $Latency_{baseline}/Latency_{UCFS}$. It is very clear that UCFS reduces the total disk latency dramatically. UCFS achieves a significant improvement over SQUID/FFS, SQUID/FFS -async and SQUIDML. UCFS improves I/O performance of proxy server by 7–19 times over SQUID/FFS, 4–7.5 times over SQUID/FFS -async and 3–9 times over SQUIDML/FFS.

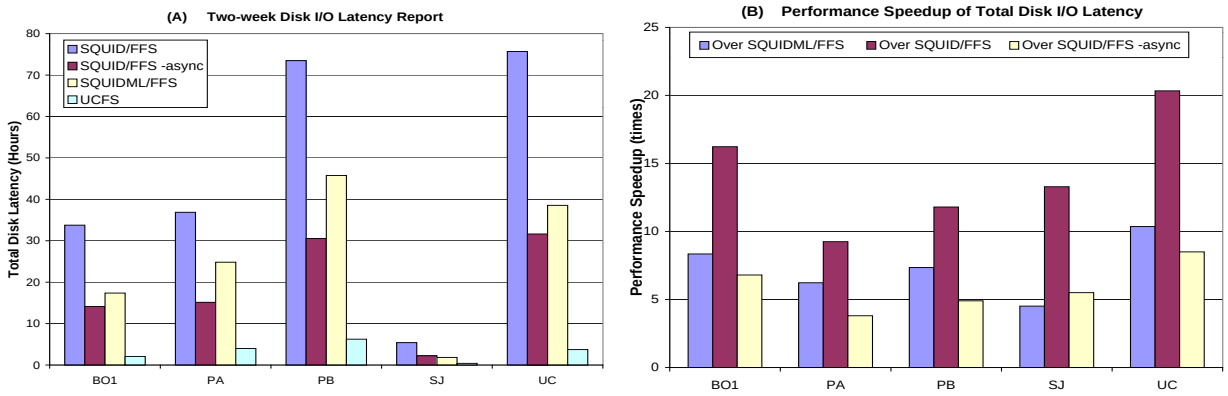

Figure 6: Total Disk I/O Latency of UCFS,SQUID/FFS,SQUID/FFS -async and SQUIDML/FFS

5.1.2 Average Disk Response Time

Figure 7(a) compares the average disk response time of UCFS and three baseline systems. Disk response time represents the total time it takes to swap in/out a file from/into disk-level cache for every URL operation. When reading a file, disk response time equals how long the proxy server must wait to get the file from the file system. When writing a file, it represents how long it takes the file system to return after a write. Figure 7(b) shows the speedups of UCFS over the three baseline systems, calculated by $Disk\ Response\ Time_{baseline}/Disk\ Response\ Time_{UCFS}$. Again, we can clearly see that UCFS achieves dramatic performance improvements over SQUID/FFS, SQUID/FFS-async and SQUIDML/FFS.

In SQUID/FFS, each URL operation spends around 30 ms on disk I/Os. SQUIDML/FFS also needs 10–20 ms on disk I/Os to serve each URL operation. Because SQUID/FFS-async avoids expensive synchronous writes, it is much faster than the above two systems, with average disk response times of about 11–14 ms. UCFS, on the other hand, needs only 1.5–3 ms on disk I/Os to handle each URL operation. This is an improvement of upto 20 times over the baseline systems.

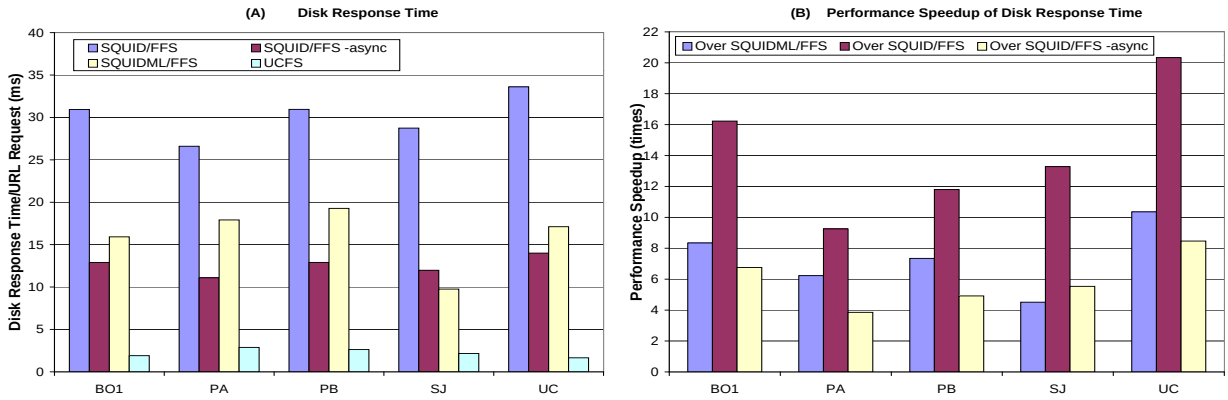

Figure 7: Average Disk Response Time of UCFS,SQUID/FFS,SQUID/FFS-async and SQUIDML/FFS

To further understand the system behaviors, we also draw the histogram of disk response time in every 1 ms range. Figure 8 shows the histogram of disk response time for the UC trace. The results of other four traces are very similar and are not shown here.

For UCFS, Figure 8(a) shows that 65.5% of the URL requests hit the RAM buffer and took less than 1 ms to finish. UCFS utilizes memory very efficiently. It uses large writes to quickly write many small files to the disk and makes buffer space available for further writes. Its large reads also effectively prefetch many files into RAM, greatly improving the read cache hit rate. About 20% of URL requests have response times higher than 6 ms, mainly because of the large disk reads and writes.

Figure 8(b) shows that for SQUID/FFS, only 10% of operations took less than 1 ms to finish, implying a much lower RAM buffer cache hit rate. Around 30% URL requests finished within 6–9 ms, which are times needed to access files on the disk. Around 60% URL requests took more than 28 ms to complete. We believe that the high costs were the results of high metadata overheads in addition to normal disk access latencies, including directory-lookups in case of DLC misses; creating new files for new objects; and expensive synchronous updates for file meta-data.

Figure 8(c) shows the results for SQUID/FFS-async. SQUID/FFS-async eliminates expensive synchronous writes for small files. As a result, more URL operations (22%) finished within 1 ms. Around 10% of requests spent more than 23 ms because of the additional expensive file meta-data reads for DLC misses. Other 60% requests spent 6–16 ms on writing/reading a file to/from the disk-level cache.

For SQUIDML/FFS, around 15% requests spent less than 1 ms disk time. Since it uses *mmap* functions to avoid many meta-data operations, SQUIDML/FFS shows better performance by "shifting" the group of slow requests in the SQUID/FFS (those higher than 28 ms) to the left by about 15 ms. Nevertheless, SQUIDML/FFS still uses small and expensive writes/reads to page data into/out of memory. Most of its operations still spent 12 ms to 24 ms time on disk response time.

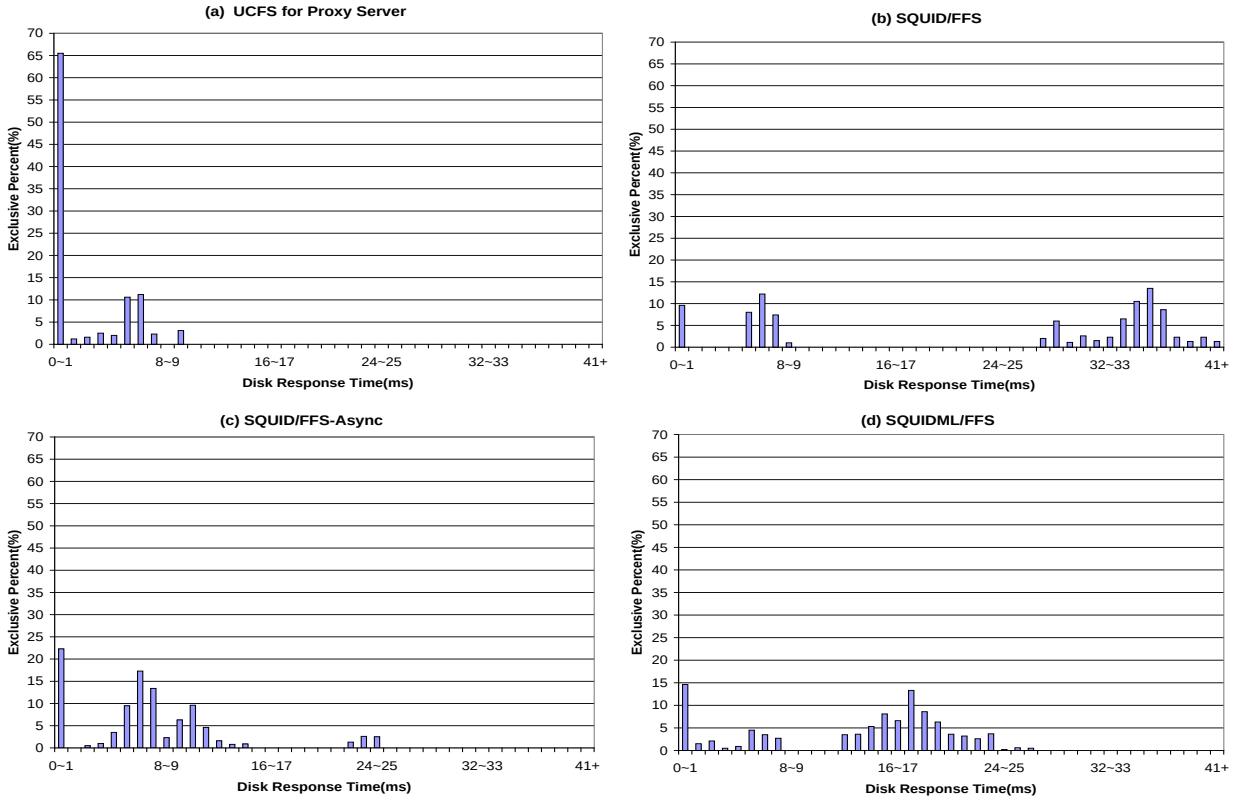

Figure 8: Histograms of Disk Response Time for UCFS, SQUID/FFS,SQUID/FFS-async and SQUIDML/FFS under UC Workload

5.1.3 Disk Reads and Writes

To gain insights on why UCFS achieve such good performance, we also categorized disk accesses into reads and writes and studied their performance separately. Figure 9 compares the total read latency and total write latency among four systems. Figure 10 shows UCFS speedup over baseline systems.

SQUID/FFS has the worst read and write performance among all systems. In SQUID, for each URL request, a cache miss will generate one file write (the proxy server retrieves the document from its original server and writes it to the disk). Each disk-hit will result in one file read from the disk. When SQUID/FFS writes a web document to the disk, it generates at least four small disk I/Os. Two writes for two-level directory inodes, one write for the file inode and one or more writes for the web object itself. When SQUID/FFS reads a web document from the disk, it also generates multiple small disk I/Os: two reads for two-level directory inodes, one read for file inode, one write to the file inode (to update the file access time), and one or more reads for the file data itself. While the FFS DLC can eliminate some reads to the inodes, the overhead is still excessively high. Moreover, the DLC hit rate is limited because a busy proxy server may have to deal with millions of files. The DLC has also to compete with other meta-data and data caches for the limited memory.

SQUIDML uses *mmapped* files to manage cached data smaller than 8 KB. It reduces (but cannot eliminate) many meta-data update overheads associated with file *open()* and *close()* operations. However, since only a small portion of all cached file data can be kept in RAM at any given moment, the system has to frequently page data in and out. This operation results in many inefficient small disk accesses. Moreover, SQUIDML uses FFS to handle files larger than 8 KB, which accounts for 30% of proxy traffic. As a result, SQUIDML/FFS has only about 50% better read performance and 30–50% better write performance than SQUID/FFS.

Since FFS-async eliminates expensive synchronous metadata updates, it can significantly improve the write performance of SQUID/FFS by 2–3 times. However, FFS-async still uses small (thus expensive) disk requests to read and write data on the disk, the disk bandwidth utilization is severely limited. Moreover, SQUID/FFS-async cannot directly improve the read performance, although it can indirectly improve read response time slightly because of the reduction of disk bandwidth competition.

UCFS, on the other hand, demonstrates drastically better performance than all the baseline systems for *both* reads and writes. For example, Figure 10 indicates that UCFS shows up to 42 times better write performance over SQUID/FFS, 9 times better write performance over SQUID/FFS-async and 7 times better read performance over SQUIDML/FFS.

As explained before, UCFS achieves such an impressive result on writes by eliminating most metadata overheads, and more importantly, by amortizing small disk writes overhead to a single large write. For example, if the average file size is 4 KB and the cluster size is 64 KB, then each cluster contains 16 files. Each file write will result in only 1/16 of a disk write. UCFS significantly improves read latencies by using large reads and efficient prefetching. Moreover, UCFS has a non-garbage CFS which has no performance degradation problem.

5.2 Document Hit Rates

We have also calculated the total Document Hit Rates (including hits in RAM and disks) for all systems. As shown in Table 3, UCFS achieves a comparable document hit rate to other two systems. This is important because UCFS achieves a good file system and I/O performance without hurting web proxy server's document hit rate. The reason that UCFS and the baseline systems have slightly different document hit rates is that they have different replacement policies, which include buffer management policies and first-timer document management algorithms.

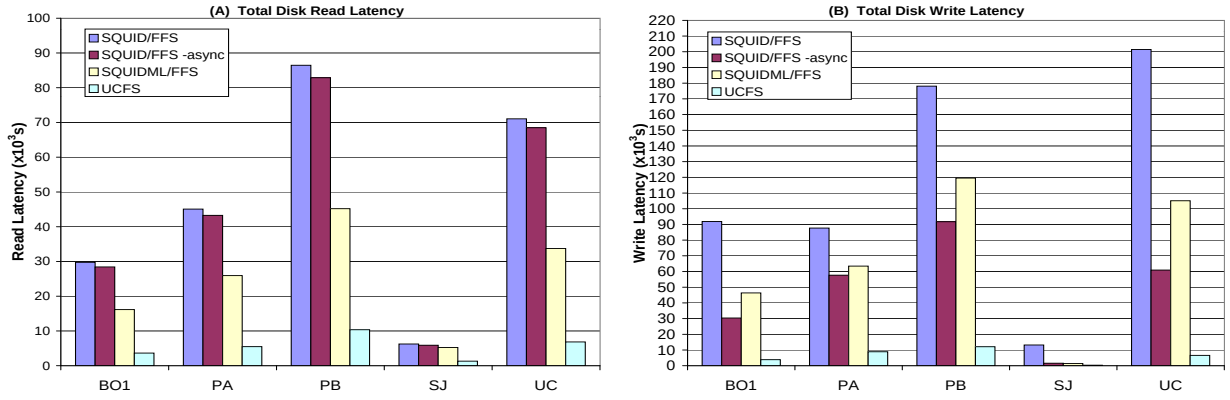

Figure 9: Total Disk Read and Write I/O Latency of UCFS,SQUID/FFS,SQUID/FFS-async and SQUIDML/FFS

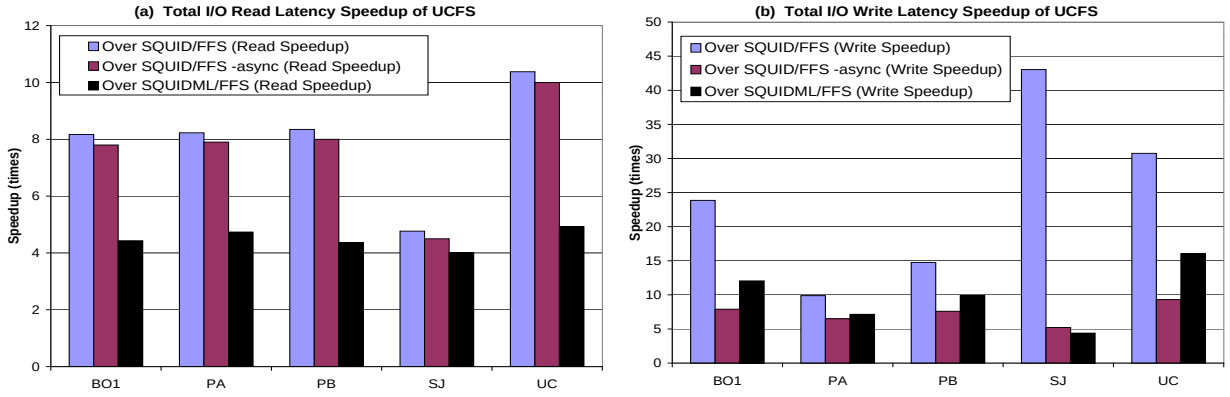

Figure 10: Read and Write Speedup

5.3 Memory Hit Rate and Prefetch Hit Rate

We have shown that UCFS improves the I/O performance of proxy servers in the following four areas: (1) Efficient in-memory data structure eliminate all meta-data I/Os; (2) the CFS design allows grouping many small writes into a few efficient large writes; (3) the CFS design always reads files in large clusters, effectively prefetching many related files into RAM with very little extra overhead; and (4) the CFS design avoids the garbage problem altogether.

One key design decision of CFS is to use large clusters as the basic read unit. Such a design allows prefetching of files and completely eliminating garbage. While avoiding garbage itself is a very important advantage, the benefit of using large clusters is largely determined by the memory hit rate — if the prefetched files in memory are not hit by the following accesses in the near future, the performance gain from prefetching is very limited. Moreover, such prefetches may even pollute the cache. A good prefetch hit rate will lead to a higher memory hit rate.

Table 4 compares the memory hit rates among UCFS, SQUID/FFS (or SQUID/FFS-async) and SQUIDML. Here the RAM size is set as 512 MB. The memory hit rates represent the percentages of all

Name	SQUID/FFS or FFS -async	SQUIDML	UCFS
BO1	44.4%	43.3%	49.8%
PA	57.7%	56.5%	51.0%
PB	57.8%	52.9%	59.2%
SJ	55.4%	56.8%	54.0%
UC	47.8%	43.0%	53.8%
Average	50.6%	50.5%	53.6%

Table 3: Document Hit Rate in Five NLANR Caches

hit documents in the RAM memory cache[4]. We can see that the UCFS has much higher memory hit rates than the baseline systems. The new scheme achieves up to 309% and 178% better memory hit rates than SQUID/FFS and SQUIDML/FFS, respectively. Table 5 shows the prefetch hit rate (as defined in Section 2.4.5 for UCFS.

The improved memory hit rate is due to successful prefetching, indicating that our locality-grouping algorithm works well.

Name	SQUID/FFS or FFS -async	SQUIDML	UCFS
BO1	6.4%	9.8%	16.1%
PA	6.1%	9.0%	15.1%
PB	5.6%	8.6%	13.8%
SJ	7.5%	13.0%	16.2%
UC	6.6%	10.5%	23.2%
Average	6.4%	10.2%	16.9%

Table 4: Memory Hit Rates of Different Architectures

Trace name	BO1	PA	PB	SJ	UC	Average
UCFS Prefetch Hit rate	30.1%	28.1%	33.8%	36.2%	38.2%	33.3%

Table 5: Prefetch Hit Rates of Different Architectures

5.4 Performance Impact of CFS Cluster Sizes

The results presented so for are for the CFS cluster size of 64 KB. We also varied the cluster size from 16 KB to 256 KB in order to find the optimal cluster size. The RAM size is fixed at 512 MB. Figure 11 shows that for the workloads we used, 64KB has the best performance. Larger clusters make it difficult to hold all similar reference locality files in one cluster, leading to wasted prefetching traffic. Smaller clusters cannot utilize the full disk bandwidth, as disk seeks and rotational latencies will increase.

5.5 Performance Impact of RAM Sizes

We have also studied the performance of the three systems under different RAM sizes. While we cannot present all the data here because of the space limitation, our observation is summarized as follows.

[4]RAM memory caches are RAM buffer caches in UCFS and I/O buffer cache for baseline systems.

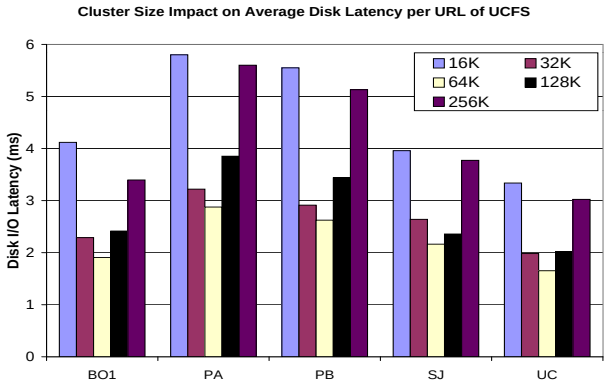

Figure 11: Average Disk I/O Latency per URL Request of UCFS with Different Cluster Size in the Two-week Period

We found that changing memory sizes has only limited impacts on proxy server performance. For example, increasing the RAM size from 512 MB to 1024 MB results in an I/O response time improvement of only 7.5% for SQUID. This is not a surprise, as previous studies have pointed out that the web cache hit rate grows in a logarithmic-like fashion with the amount of traffic and the size of the client population[4, 5].

UCFS performs slightly better when the memory size increases. The speedup of UCFS over SQUID/FFS increases slightly when the memory size increases, as shown in Figure 12. We believe that this is because the locality-based grouping algorithm has better chances to find file candidates when the buffer size increases.

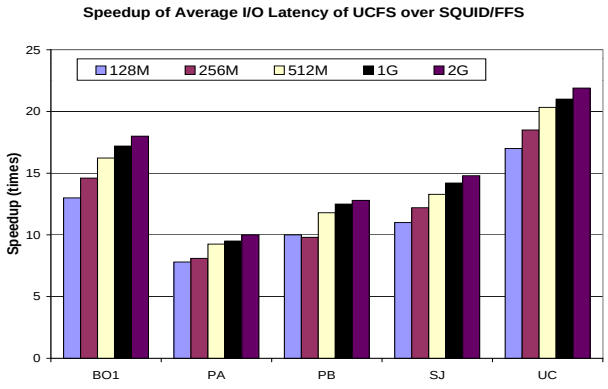

Figure 12: I/O Response Time Speedup under Different Memory Sizes

6 Related Work

- **Unix File System**
  Many papers tried to improve Unix file system performance. Sprite LFS [10] and BSD LFS [11] were developed to solve the small writes problem of FFS. So far there are no major commercial operating systems supporting it. Ganger *et al.* proposed a solution called Soft-Updates to enhance the performance of FFS by using safe asynchronous writes [14, 15]. Ganger and Kaashoek improved

FFS disk bandwidth for small files by embedded inodes and explicit grouping [31]. While progress has been made, current general-purpose file systems still have poor performance for proxy server workloads.

- **General Issues of Web Proxy Servers**
  Large Web Proxy Server is a very popular research topic at present. A number of papers show that web proxy traces have an excellent long-term locality [25, 26]. A recent evaluation showed that document hit ratios ranged from 32% to 58% within 29 large commercial web proxy servers [6]. Many other research papers also reported similar hit rates in web proxy servers [32, 2].

- **Web Prefetching in Web Proxy Servers**
  Wcol [33] is a proxy software that parse HTML and prefetch documents, links and embedded images. Kaashoek *et al*. [34] suggested a web prefetching technique to fetch in-lined images and places such files adjacent to the HTML files and reads them as a unit (effectively prefetching the images).

- **Piggyback Cache Validation**
  Krishnamurthy and Wills proposed piggyback invalidation policies [35, 36]. Their piggyback cache validation (PCV) and piggyback server invalidation (PSI) tried to reduce the cache validation traffic between proxy caches and servers. The two ideas both use the original web server's information to partition a group of affiliated (or related) web documents. Proxy caches and web servers always exchange or check the validate information of this group together. Since these documents of the same group are likely to be validated simutaneously, these schemes reduce the average networking cost and staleness ratio.

  Because UCFS groups affiliated documents together, we can apply the piggyback invalidation techniques to UCFS to reduce networking cost and maintain strong coherency. When requesting/revalidating one web document from a web server, UCFS can piggyback validation requests for other files in the same cluster.

- **I/O and File Systems in Web Proxy Servers**
  Several studies suggested that disk I/O traffic was a significant overhead on Web proxy servers [37, 38]. Rousskov and Soloviev observed that disk delays contribute 30% to the total hit response time [38]. Their experiments showed that on three different level proxy caches using SQUID 1.1, disk response times mostly ranged from 20 ms to 100 ms. This result is similar to the values observed in our SQUID/FFS baseline system. Soloviev and Yahin suggested that proxies use multiple disks to balance the load and reduce long seeks by disk partitions [39]. Maltzahn *et al.* [3] proposed two methods to reduce the overhead for proxies: maintaining the same directory to store web documents from the same host and using *mmap* file in virtual memory to handle small objects so as to reduce meta-data operations. Markatos et al proposed a similar method to improve file system performance, storing objects of similar sizes in the same directory (called BUDDY). They also developed a technique using large continuous disk writes (called STREAM) to reduce disk latency [17]. By calculating the average disk latency for URL operation in the simulation, they concluded SQUID spent around 50 ms latency for every URL operation with a 512 MB I/O buffer cache. This result is also similar to the values obtained in our SQUID/FFS baseline system. Iyengar et al. presented several disk storage allocation algorithms for managing persistent storage which minimize the number of disk seeks for both allocations and deallocations [40]. They claimed their algorithms can result in considerable performance improvements over database and file systems for web-related workloads.

  But the I/O performance improvement for large proxy servers is still limited in above methods. They all tried to improve I/O performance on top of current file system which can only get quite limited effect. We build our own simple but highly efficient customized file system on a raw disk. Compared

to the above methods, we save most meta-data's I/O overheads and maximize the utilization of disk bandwidth. Our user-space file system is combined well with disk I/O subsystem in such a web proxy server related load.

- **Specialized File Systems**
  There are many commercial specific systems developed for the web proxy server market. Network Appliance presented $NetCache^{TM}$ 4.1 which provides Web proxy and cache service to Web clients that support the use of proxy agents [41]. CacheFlow improved I/O performance by building a special operating system with an application-specific file system executing on closed hardware [42]. Moreover, no design details are available. Multimedia storage servers provide guaranteed Quality of Service for reading and writing streaming media by implementing specialized data placement and request scheduling algorithms to provide those services [43].

7 Conclusions

This paper presents a novel user-space, customized file system called UCFS for high-performance web proxy servers. The system has the following significant advantages over traditional approaches:

1. UCFS manages file data and meta-data on a raw-disk or disk partition. Without the limitation of an existing general-purpose file system, we are free to design efficient algorithms that are optimized for web proxy cache applications. Moreover, since UCFS runs completely in the user-space, it is easy and inexpensive to implement and maintain. These is no need to modify the OS kernel.

2. The design has excellent portability, since it runs as a part of a regular, user-space program. UCFS does not need special support from the OS to achieve high I/O performance.

3. The Cluster-structured File System (CFS) design, together with the algorithms which group documents with similar access locality together in the same cluster, significantly improve *both* read and write performance.

4. Unlike other log-structured approaches, *there is no garbage collection in CFS*, as CFS does not generate any garbage at all.

5. The efficient in-memory file lookup tables completely eliminate the overhead caused by meta-data I/Os. The system also avoids the overhead of periodically flushing dirty data to the disk.

Extensive simulation confirms that UCFS can drastically improve proxy server performance. By running five two-week long NLANR traces, we found that UCFS reduces SQUID/FFS's disk I/Os by up to 93%. It achieves up to *19 times* better I/O performance than SQUID/FFS, *7.5 times* better than SQUID/FFS -async and *10 times* better performance than SQUIDML/FFS, in terms of disk response time.

It should be pointed out that the traces we used in this research are root-level traces. Traces from lower-level caches such as institution leaf-level proxy caches normally have better locality [7, 23]. Therefore we believe that our scheme will have even better performance for lower-level proxy servers, since the locality-based grouping algorithm will work better.

8 Future Work

We believe that UCFS can also be used to improve the performance of many other network applications, such as Web servers and file caching servers. I/Os are the major bottleneck of these applications. For

example, a UCFS system can co-exist with a regular file system to accelerate a web server. Users maintain their web documents in the reliable and familiar regular file system. When the web server accesses a web page the first time, the file is copied to the UCFS system and all future references to the file go to the UCFS, until the file is updated. By doing so, we can dramatically improve the web server performance, while all the operations are transparent to users.

For future work, we are trying to improve the locality-grouping algorithms in order to further improve the prefetch hit rates. We will also try to actually implement UCFS and test its performance in a real world environment. Finally, we will study issues of applying UCFS to other Web-related platforms such as Web servers.

Acknowledgments

This work is supported in part by the National Science Foundation Career Award CCR-9984852, and an Ohio Board of Regents Computer Science Collaboration Grant.

References

[1] J. Wang, R. Min, Z. Wu, and Y. Hu, "Boosting I/O performance of internet servers with user-level custom file systems," in *Proceedings of the 2nd Workshop of Performance and Architecture of Web Servers (PAWS2001). Published as a special issus (Vol. 29, No. 2) of ACM Sigmetrics Performance Evaluation Reviews*, (Boston, MA), pp. 26–31, Feb. 2001.

[2] J. Almeida and P. Cao, "Measuring proxy performance with the wisconsin proxy benchmark," Tech. Rep. 1373, Computer Science Department,University of Wisconsin-Madison, April 1998.

[3] C. Maltzahn, K. J. Richardson, and D. Grunwald, "Reducing the disk I/O of web proxy server caches," in *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, (Berkeley, CA), pp. 225–238, USENIX Association, June 6–11 1999.

[4] S. D. Gribble and E. A. Brewer, "System design issues for Internet middleware services: Deductions from a large client trace," in *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, (Monterey, CA), Dec. 1997.

[5] B. M. Duska, D. Marwood, and M. J. Freeley, "The measured access characteristics of World-Wide-Web client proxy caches," in *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, (Monterey, CA), Dec. 1997.

[6] A. Rousskov and D. Wessels, "The Third Cache-off the Offical Report," Oct 2000. The Measurement Factory, Inc.

[7] A. Rousskov and V. Soloviev, "A performance study of the squid proxy on http," in *World-Wide Web Journal , Special Edition on WWW Characterization and Performance Evaluation, 1999.*, 1999.

[8] D. Wessels, "SQUID frequently asked questions," 2001. http://www.squid-cache.org/Doc/FAQ/FAQ.html.toc13.

[9] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems," tech. rep., Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkeley, Oct. 1988.

[10] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 26 – 52, Feb. 1992.

[11] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *Proceedings of Winter 1993 USENIX*, (San Diego, CA), pp. 307–326, Jan. 1993.

[12] Y. Hu and Q. Yang, "DCD—disk caching disk: A new approach for boosting I/O performance," in *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)*, (Philadelphia, Pennsylvania), pp. 169–178, May 1996.

[13] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, (Austin, TX), pp. 155–162, 8-11 Nov. 1987. In *ACM Operating Systems Review 21*:5, 1987.

[14] G. R. Ganger and Y. N. Patt, "Metadata update performance in file systems," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 49–60, Nov. 1994.

[15] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: a solution to the metadata update problem in file systems," *ACM Transactions on Computer Systems*, vol. 18, pp. 127–153, May 2000.

[16] C. R.Lumb, J. Schindler, G. R. Ganger, and D. F. Nagle, "Towards higher disk head utilization: Extracting free bandwidth from busy disk drives," in *Proceedings of the 2000 Conference on Operating System Design and Implementation (OSDI)*, (San Diego), Oct. 2000.

[17] E. P. Markatos, M. G. H. Katevenis, D. Pnevmatikatos, and M. Flouris, "Secondary storage management for web proxies," in *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS-99)*, (Berkeley, CA), pp. 93–104, USENIX Association, Oct. 11–14 1999.

[18] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File system logging versus clustering: A performance comparison," in *Proceedings of 1995 USENIX*, (New Orleans, LA), pp. 249–264, Jan. 1995.

[19] T. Nightingale, Y. Hu, and Q. Yang, "The design and implementation of DCD device driver for UNIX," in *Proceedings of the 1999 USENIX Technical Conference*, (Monterey, California), pp. 295–308, Jan. 1999.

[20] U. Vahalia, *UNIX Internals — The New Frontiers*. Prentice Hall, 1996.

[21] R. L. Rivest, "MD5 unoffical homepage." http://userpages.umbc.edu/ mabzug1/cs/md5/md5.html, 1991.

[22] B. Jenkins, "A new hash functions for hash table lookup," *Dr. Dobb's Journal*, Sept 1997.

[23] A. Mahanti, C. Williamson, and D. Eager, "Traffic analysis of a web proxy caching hierarchy," *IEEE Network*, vol. 14, pp. 16–23, May/June 2000.

[24] M. Arlitt and C. Williamson, "Trace-driven simulation of document caching strategies for Internet web servers," *Simulation Journal*, vol. 68, pp. 23–33, Jan 1997.

[25] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proceedings of the INFOCOM '99 conference*, Mar. 1999.

[26] P. Barford, A.Bestavros, A.Bradley, and M.E.Crovella, "Changes in web client access patterns: characteristics and caching implications," *World Wide Web*, vol. 2, pp. 15–28, January 1999.

[27] G. R. Ganger and Y. N. Patt, "Using system-level models to evaluate I/O subsystem designs," *IEEE Transactions on Computers*, vol. 47, pp. 667–678, June 1998.

[28] Quantum, 2000. Quantum Atlas[tm] 10KII disk drives.

[29] D. Wessels, "NLANR CACHE README," April 1998. ftp://ircache.nlanr.net/Traces/README.

[30] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1." http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc2616.html, June 1999.

[31] G. R. Ganger and M. F. Kaashoek., "Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files," in *Proceedings of the USENIX Technical Conference, January 1997*, pp. 1–17, Jan 1997.

[32] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox, "Removal policies in network caches for world-wide web documents," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, vol. 26,4 of *ACM SIGCOMM Computer Communication Review*, (New York), pp. 293–305, ACM Press, Aug. 26–30 1996.

[33] Wcol Group, "WWW collector: the prefetching proxy server for WWW," 1997. http://shika.aistnara.ac.jp/products/wcol/wcol.html.

[34] M. F. Kaashoek, D. R. Engler, G. R. G. anger, and D. A. Wallach, "Server operating systems," in *1996 SIGOPS European Workshop*, Sept. 1996.

[35] B. Krishnamurthy and C. E. Wills, "Study of piggyback cache validation for proxy caches in the World Wide Web," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems (ITS-97)*, (Berkeley), pp. 1–12, USENIX Association, Dec. 8–11 1997.

[36] B. Krishnamurthy and C. E. Wills, "Piggyback server invalidation for proxy cache coherency," in *Proceedings of the 7th International WWW Conference*, (Brisbane, Australia), Apr. 1998.

[37] K. Kant and P. Mohapatra, "Scalable Internet servers: issues and challenges," in *Performance Evaluation Review*, 2000.

[38] A. Rousskov and V. Soloviev, "On peformance of caching proxies," in *Proceedings of SIGMETRICS1998/PERFORMANCE1998*, (Madison, WI), pp. 272–273, Jun 1998.

[39] V. Soloviev and A.Yahin, "File placement in a web cache server," in *Proceedings of 10-th ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[40] A. Iyengar, S. Jin, and J. Challenger, "Efficient algorithms for persistent storage allocation," in *Proceedings of the 18th of IEEE Symposium on Mass Storage Systems*, (San Diego, California), April 2001.

[41] P. Danzig, "Netcache architecture and deployment," 1998. http://www.netapp.com/technology/level3/3029.html.

[42] CacheFlow, "High-performance web caching white paper," 1998.

[43] C. Martin, P. S. Narayanan, B. O. R. Rastogi, and A. Silberschatz, *The Fellini multimedia storage server, Chapter 5*. Kluwer Academic Publishers, 1996.