Storage management for web proxies

Elizabeth Shriver
Bell Laboratories
shriver@bell-labs.com

Eran Gabber
Bell Laboratories
eran@bell-labs.com

Lan Huang
SUNY Stony Brook
lanhuang@cs.sunysb.edu

Christopher A. Stein
Harvard University
stein@eecs.harvard.edu

Abstract

Today, caching web proxies use general-purpose file systems to store web objects. Proxies, e.g., Squid or Apache, when running on a UNIX system, typically use the standard UNIX file system (UFS) for this purpose. UFS was designed for research and engineering environments, which have different characteristics from that of a caching web proxy. Some of the differences are high temporal locality, relaxed persistence requirements, and a different read/write ratio. In this paper, we characterize the web proxy workload, describe the design of Hummingbird, a light-weight file system for web proxies, and present performance measurements of Hummingbird. Hummingbird has two distinguishing features: it separates object naming and storage locality through direct application-provided hints, and its clients are compiled with a linked library interface for memory sharing. When we simulated the Squid proxy, Hummingbird achieves document request throughput 2.3–9.4 times larger than with several different versions of UFS. Our experimental results are verified within the Polygraph proxy benchmarking environment.

1 Introduction

Caching web proxies are computer systems dedicated to caching and delivering web content. Typically, they exist on a corporate firewall or at the point where an Internet Service Provider (ISP) peers with its network access provider. From a web performance and scalability point of view, these systems have three purposes: improve web client latency, drive down the ISP's network access costs because of reduced bandwidth requirements, and reduce request load on origin servers.

Squid and Apache are two popular web proxies. Both of these systems use the standard file system services provided by the host operating system. On UNIX this is usually UFS, a descendant of the 4.2BSD UNIX Fast File System (FFS) [13]. FFS was designed for workstation workloads and is not optimized for the different workload and requirements of a web proxy. It has been observed that file system latency is a key component in the latency observed by web clients [21].

Some commercial vendors have improved I/O performance by rebuilding the entire system stack: a special operating system with an application-specific file system executing on dedicated hardware (e.g., CacheFlow, Network Appliance). Needless to say, these solutions are expensive. We believe that a lightweight and portable file system can be built that will allow proxies to achieve performance close to that of a specialized system on commodity hardware, within a general-purpose operating system, and with minimal changes to their source code; Gabber and Shriver [5] discuss this view in detail.

We have built a simple, lightweight file system library named Hummingbird that runs on top of a raw disk partition. This system is easily portable—we have run it with minimal changes on FreeBSD, IRIX, Solaris, and Linux. In this paper we describe the design, interface, and implementation of this system along with some experimental results that compare the performance of our system with a UNIX file system. Our results indicate that Hummingbird's throughput is 2.3–4.0 times larger than a simulated version of Squid running UFS mounted asynchronously on FreeBSD, 5.4–9.4 times faster than Squid running UFS mounted synchronously on FreeBSD, 5.6–8.4 times larger than a simulated version of Squid running UFS with soft updates on FreeBSD, and 5.4–13 times larger than XFS and

EFS on IRIX (see Section 4). We also performed experiments using the Polygraph environment [18] with an Apache proxy; the mean response time for hits in the proxy is 14 times smaller with Hummingbird than with UFS (see Section 5).

Throughout the rest of this paper, we use the terms *proxy* or *web proxy* to mean *caching web proxy*. Section 2 presents the important characteristics of the proxy workload considered for our file system. It also presents some background on file systems and proxies that is important because it motivates much of our design. Section 3 describes the Hummingbird file system. Our experiments and results are presented in Sections 4 and 5. Section 6 discusses related work in file systems and web proxy caching.

2 File system limitations

The web proxy workload is different than the standard UNIX workload; these differences are discussed in Section 2.1. Due to these differences, the performance which can be attained for a web proxy running on UFS is limited by some of the features; these limitations are discussed in Section 2.2.

2.1 Web proxy workload characteristics

Web proxy workloads have special characteristics, which are different from those of a traditional UNIX file system workload. This section describes the special characteristics of proxy workloads.

We studied a week's worth of web proxy logs from a major, national ISP, collected from January 30 to February 5, 1999. This proxy ran Netscape Enterprise server proxy software and the logs were generated in Netscape-Extended2 format. For the purpose of our analysis we isolated the request stream to those that would affect the file system underlying the proxy. Thus we excluded 34% of the GET requests which are considered non-cacheable by the proxy. If we could not find the file size, we removed the log event. This preprocessing results in the removal of about 4% of the log records, nearly all during the first few days. We eliminated the first few days and are left with 4 days of processed logs containing 4.8 million requests for 14.3 GB of unique cacheable data and 27.6 GB total requested cacheable data.

Characteristics of a web proxy and its workload are:

Persistence. A web proxy only writes files retrieved from an origin server. Thus, these are just cached files, and can be retrieved again if necessary.

Naming. The web proxy application determines the name of a file to be written into the file system. In a traditional UNIX file system, file names are normally selected by a user.

Reference locality. Client web accesses are characterized by a request for an HTML page followed by requests for embedded images. The set of images within a cacheable HTML page changes slowly over time. So, if a page is requested by a client, it makes sense for the proxy to anticipate future requests by prefetching its historically associated images.

We studied one day of the web proxy log for this reference locality. The first time we see an HTML file, we coalesce it with all following non-HTMLs from the same client to form the primordial *locality set*, which does not change. The next time the HTML file is referenced, we form another locality set in the same manner and compare its file members with those of the primordial locality set. The average hit rate (the ratio between the size of the latter sets and the size of the primordial set) across all references is 47%. Thus, on average, a locality set re-reference accesses almost half of the files of the original reference. One of the reasons that this hit rate is small might be due to the assumption that all non-HTML files that follow a HTML file are in the same locality set; this is clearly not true if a user has multiple active browser sessions. Also, we determined the type of file using the file extension, thus possibly placing some HTML files in another file's locality set. We also studied the size of the locality sets in bytes, and found that 42% are 32 KB or smaller, 62% are 64 KB or smaller, 73% are 96 KB or smaller, and 80% are 128 KB or smaller.

File access. Several older UNIX file system performance studies have shown that files are usually accessed sequentially [16, 1]. A recent study [19] has suggested that this is changing, especially for memory-mapped and large files. However, UNIX file systems have been designed for the traditional sequential workload. Web proxies have an even stronger and more predictable pattern of behavior. Web proxies currently always access files sequentially and in their entirety. (This may change when range requests are more popular.)

File size. Most cacheable web documents are small; the median size of requested files is 1986 B and the average size is 6167 B. Over 90% of the references are for files smaller than 8 KB.

Idleness. The proxy workload is characterized by a large variability in request rate and frequent idle periods. Using simulation, we found, in fact, in each 1-minute interval, the disk is idle 57–99% of the duration of the interval, with a median of 82%. If the request rate in the trace doubles, the disk is idle less of the time, with a median of 60%. In the busiest periods, it is idle only 1%. Note that when the request rate doubles, the disk is almost saturated in the busy intervals (idle time drops to 1%), while the disk remains idle in the low-activity intervals. This idleness study used assumptions that represent Hummingbird: the disk has 64 KB blocks, all new files are written to the disk, and only data needs to be written to disk (no meta-information). The existence of idle periods is crucial for the design of Hummingbird since Hummingbird performs background bookkeeping operations in those idle periods (see Section 3.5).

2.2 UFS performance limitations

Due to the proxy workload characteristics presented in the previous section, UFS has a number of features which are not needed, or could be greatly simplified, so that file system performance could be improved. Table 1 presents the features on which UFS and a desired caching web proxy file system should differ. We now discuss these features.

UFS files are collections of fixed-size blocks, typically 8 KB in size. When accessing a file, the disk delays are due to disk head positioning occurring when the file blocks are not stored contiguously on disk. UFS attempts to minimize the disk head positioning time by storing the file blocks contiguously and prefetching blocks when a file is accessed sequentially, and does a good job of this for small files. Thus, when the workload consists of mostly small files, the largest component of disk delays are due to the reference stream locality not corresponding with the on-disk layout. UFS attempts to reduce this delay by having the user place files into directories, and locates files in a directory on a group of contiguous disk blocks called cylinder groups. Thus, reference locality is tied to naming. Here, the responsibility for performance lies with the application or user, who must construct a hierarchy with directory locality that matches future usage patterns. In addition, to reduce file lookup times, the directory hierarchy must be well-balanced and any single directory should not have too many entries. Squid attempts to balance the directory hierarchy, but in the process distributes the reference stream across directories, thus destroying locality. Apache maps files from the same origin server into the same directory. For specifying locality by a web proxy, a more direct and low-overhead mechanism can be used.

Experience with Squid and Apache has shown that it is difficult for web proxies to use directory hierarchies to their advantage [11]. Deep pathnames mean long traversal times. Populated directories slow down lookup further because many legacy implementations still do a linear search for the filename through the directory contents. The hierarchical name space allows files to be organized by separating them across directories, but this is not needed by a proxy. What the proxy actually needs is a flat name space and the ability to specify storage locality.

UFS file meta-data is stored in the i-node, which is updated using synchronous disk writes to ensure meta-data integrity. A caching web proxy does not require file durability for correctness, so it is free to replace synchronous meta-data writes with asynchronous writes to improve the performance (which is done with soft updates [6]) and eliminate the need to maintain much of the meta-data associated with persistence.

Traditional file systems also force two architectural issues. First, the standard file system interface copies from kernel VM into the applications' address space. Second, the file system caches file blocks in its own buffer cache. Web proxies manage their own application-level VM caches to eliminate memory copies and use private information to facilitate more effective cache management. However, web documents cached at the application level are likely to also exist in the file system buffer cache, especially if recently accessed from disk. This multiple buffering reduces the effective size of the memory. A single unified cache solves the multiple buffering and configuration problems. Memory copy costs can be alleviated by passing data by reference rather than copying. Both of these can be done using a file system implemented by a library that accesses the raw disk partition. Another approach for alleviating multiple buffering is using memory-mapped files as done by Maltzahn et al. [11].

3 File system design

The design of Hummingbird is influenced by the proxy workload characteristics discussed in Section 2. Hummingbird uses locality hints generated by the proxy to pack files into large, fixed-size extents called *clusters*, which are the unit of disk access. Therefore, reads and writes are large, amortiz-

Table 1: Comparison of file system features.

feature	UFS	Hummingbird
name space	hierarchical name space	flat name space
reference locality	application manages directories	application sends locality hints
file meta-data	meta-data kept on disk; synchronous updates preserve consistency	most meta-data in memory
disk layout of files	data in blocks, with separate i-node for meta-data	file and meta-data stored contiguously
disk access	could be as small as a block size	cluster size (typically 32 or 64 KB)
interface	buffers are passed; memory copies are necessary	pointers are passed

ing disk positioning times and interrupt processing.

Hummingbird manages a large memory cache. Since Hummingbird is implemented by a library that is linked in with the proxy, no memory copies or memory mappings are required to move data from the file system to the proxy. The file system simply passes a pointer. Likewise, the proxy passes the file system a pointer when it writes data. We have only designed the file system for a single client, so protection is not necessary. Since the client (the proxy) handles all data transfers to and from the system, it must be trusted in any case. The proxy may be multi-threaded or have multiple processes[1], in which case access is serialized with a single lock on the file system meta-data that is released before blocking for disk I/O. Using a single lock may slow the system under a heavy load. Since the file system is I/O bound, the lock is held only when referring to Hummingbird's data structures. The lock is released before a thread blocks for disk I/O. No locking is needed when accessing file data.

Since the typical workload is bursty, Hummingbird is designed to reduce the response time during bursty periods, and perform maintenance activities during the idle periods present in the workload. Hummingbird performs the maintenance activities by calling several daemons responsible for: (1) reclaiming main memory space by writing files into clusters, and (2) reclaiming disk space by deleting unused clusters.

While there are invariants across proxy workloads, some characteristics will change. We have designed Hummingbird to be configurable so that the system can be optimized for a proxy workload and the underlying storage hardware. To this effect, Hummingbird has several parameters that the proxy is free to set to optimize the system for its workload. The parameters set at file system initialization include: size of a cluster, memory cache eviction policy, file hash table size, file and cluster lifetimes, disk data layout policy, and recovery policies.

[1]To perform memory management for the multi-process version of Hummingbird, the processes have a shared memory region which is used for `malloc()`.

3.1 Hummingbird objects

Hummingbird stores two main types of objects in main memory: *files* and *clusters*. A file is created by a `write_file()` call. Clusters contain files and some file meta-data. Grouping files into clusters allows the file system to physically collocate files together, since when a cluster is read from disk, all of the files contained in the cluster are read. Clusters are *clean*, i.e., they can be evicted from main memory by reclaiming their space without writing to disk, since a cluster is written to disk as soon as it is created. (Section 3.5 discusses where on disk the clusters are written.)

The application provides locality hints by the `collocate_files(fnameA, fnameB)` call. The file system saves these hints until the files are assigned to clusters. This assignment occurs as late as possible, that is, when space is needed in main memory. At this point, the file system attempts to write `fnameA` and `fnameB` in the same cluster. It is possible for a file to be a member of multiple clusters, and stored in multiple locations on disk by the application sending multiple hints (e.g., `collocate_files(fnameA, fnameB)` and `collocate_files(fnameC, fnameB)`). For proxy caches, this is a useful feature since embedded images are frequently referenced in a number of related HTML pages.

When the file system is building a cluster, it determines which files to add to the cluster using an LRU ordering according to the last time the file was read. If the least-recently-used file has a list of collocated files, then these files are added to the cluster if they are in main memory. (If a file is on the collocation list, and already has been added to a cluster, it can still be added to the current cluster if the file is in memory.) Files are packed into the cluster until the cluster size threshold is reached, or until all files on the LRU list have been processed. This way, small locality sets with similar last-read times can be packed into the same cluster. Another possible algorithm to pack files into clusters is the Greedy Dual Size algorithm [3].

Large files are special. They account for a very small fraction of the requests, but a significant fraction of the bytes transferred. In the log we analyzed, files over 1 MB accounted for over 8% of the bytes transferred, but only 0.02% of the requests. Caching these large files is not important for the average latency perceived by clients, but is an important factor in the network access costs of the ISP. It is better to store these large files on disk, and not in the file system cache in memory. The `write_nomem_file()` call bypasses the main memory cache and writes a file directly to disk; if the file is larger than the cluster size, multiple clusters are allocated. Having an explicit `write_nomem_file()` function allows the application to request that any file can bypass main memory, not just large files.

3.2 Meta-data

Hummingbird maintains three types of meta-data: file system meta-data, file meta-data, and cluster meta-data.

File system meta-data. To determine when main memory space needs to be freed, Hummingbird maintains counts of the amount of space used for storing the files and the file system data. To assist with determining which files and clusters to evict from main memory, Hummingbird maintains two LRU lists, one for files which have not yet been packed into clusters and another for clusters that are in memory.

File meta-data. A hash table stores pointers to the file information such as the file number (discussed below), status, and a reference count of the number of users that are currently reading the file. The file status field identifies whether the file is not in a cluster, in one cluster, in multiple clusters, or not cacheable. Until a file becomes a member of a cluster, the file name and file size need to be maintained as part of the file meta-data. We also maintain a list of files that should be collocated with this file. When a file is added to a cluster, the file meta-data must include the cluster ID and the file reference count for that file.

It is natural for a proxy to use the URL as a file name. URLs may be extremely long, and since we have many small files, the file names may take up a large portion of main memory if they were kept permanently in memory. Thus, we save the file name with the file data in its cluster and not permanently in memory. Internally, Hummingbird hashes the file name into a 32-bit index, which is used to locate the file meta-data. Hash collisions can be detected by comparing the requested file name with the file name stored in the cluster. If there is a collision, the next element in the hash table bucket is checked.

Cluster meta-data. A cluster table contains information about each cluster on disk: the status, last-time accessed, and a linked list of the files in the cluster. The cluster status field identifies whether the cluster is empty, on disk, or in memory. For our file system, the cluster ID identifies the location of the cluster on disk. While a cluster is in memory, the address of the cluster in memory is needed. The last-time accessed is needed to determine the amount of time since the cluster was last touched.

3.3 The Hummingbird interface

This section describes the basic Hummingbird calls. All routines return a positive integer or zero if the operation succeeds; a negative return value is an error code.

- `int write_file(char* fname, void* buf, size_t sz);` This function writes the contents of the memory area starting at `buf` with size `sz` to the file system with filename `fname`. It returns the size of the file. Once the pointer `buf` is handed over to the file system, the application **should not** use it again. Hummingbird will eventually pack the file into a cluster, free the buffer, and write the cluster to stable storage.
- `int read_file(char* fname, void** buf);` This function sets `*buf` to the beginning of the memory area containing the contents of the file `fname`. It increments a reference count and returns the size of the file. If the file is not in main memory, another file or files may be evicted to make room, and a cluster containing the specified file will be read from disk.
- `int done_read_file(char* fname, void* buf);` This function releases the space occupied by the file in main memory by decrementing the reference count; the application **should not** use the pointer again. Every `read_file()` must be accompanied

by a `done_read_file()`. Otherwise, the file will stay in memory indefinitely (until the application program terminates).

- `int delete_file(char* fname);` This function deletes the file `fname`. An error code is returned if the file has any active `read_file()`'s.
- `int collocate_files(char* fnameA, char* fnameB);` This function attempts to collocate file `fnameB` with file `fnameA` on disk. Both files must be previously written (by calling `write_file()`).
- `int write_nomem_file(char* fname, void* buf, size_t sz);` This function bypasses the main memory cache and writes a file directly to disk. This file is flagged so that when it is read, it does not compete with other documents for cache space and is immediately released after the application issues the `done_read_file()`.

Missing from this API are commands such as `ls` and `df`. We have seen no need for such commands for a caching web proxy. For example, the existence of a file can be determined with `read_file()`.

3.4 Recovery

Web proxies are slowly convergent; it takes days to reach the maximal hit rate. Consequently, proxy cache contents must be saved across system failures. At the same time, recovery must be quick. With today's disk sizes, the system cannot wait for full disk scans before servicing document requests. Hummingbird warms the main memory cache with files and clusters that were "hot" before the system reboot. Bounding file create and delete persistence rather than attaching them to system call semantics allows for higher performance.

Data stored on disk. Hummingbird's disk storage is segmented into four regions:

- clusters, which store the file data and meta-data,
- mappings of files to clusters, which allow a file to be quickly located on disk by identifying which clusters the file is in,
- the hot cluster log, which caches frequently used clusters, and
- the delete log, which stores small records describing intentional deletes.

The mappings of files to clusters is part of the file meta-data described in Section 3.2.

Warming the cache. Hummingbird lacks a directory structure and all meta-data consistency dependencies are contained within clusters. There is no need for an analog of the UNIX `fsck` utility to ensure file system consistency after a crash. During a planned shutdown, Hummingbird will write the file-to-cluster mappings to disk. (They are also written periodically.) During a crash, the system has no such luxury. So, while the file system is guaranteed to be consistent with itself, the lack of directories might make it impossible to locate files on disk immediately after a crash if the file-to-cluster mapping was not up to date. Hummingbird speeds up crash recovery time with a log containing the cluster identifiers of popular clusters. The `sync_hot_clusters_daemon()` creates this log using the cluster LRU list and writes it to disk periodically. After a crash, these clusters are scanned in first, quickly achieving a hit rate close to that before the crash.

Persistence. Hummingbird does not change disk contents immediately for file and cluster deletions. Research with journalling file systems has shown that hard meta-data update persistence is expensive, due to the necessity for updating stable storage synchronously [22]. Hummingbird does not provide hard persistence, but uses a log of intentional deletions to bound the persistence of deletions. Records describing deletions, either cluster or file, are written into the log, which is buffered in memory. Periodically, the log is written out to disk according to the specified thresholds. The user specifies when the log should be written by specifying either a number of files threshold (i.e., once X files have been recorded in the log, it must be written to disk), or a threshold on the passing of time (i.e., the log must be written to disk at least once every Y seconds).

The log is structured as a table, indexed by cluster or file identifier. When a file or cluster is overwritten on disk, the delete intent record is removed from the log. Records contain file or cluster identifiers as well as a generation number, which is stored in the on-disk meta-data and used to eliminate the possibility of replayed deletes.

The recovery procedure. During recovery, all four regions of the disk are locked exclusively by the recovery process. The Hummingbird interface comes alive to the application early in the process – after the hot cluster log of the previous session has been recovered. However, Hummingbird will not write new files or clusters to disk until recovery has completed; `write_file()` calls will fail. It continues to recover the clusters in the background and rebuild the in-memory meta-data. During this phase, requests that do not hit in the currently-rebuilt meta-data are checked in the file-to-cluster mappings to identify the cluster that needs to be recovered. Since the file-to-clusters mapping can be out-of-date (due to a crash), a file without a file-to-cluster mapping is viewed as not in the file system.

We now outline the sequence of events during crash recovery; recovery is much simpler during a planned shutdown. (1) Crash or power failure. The system reboots and enters recovery mode. (2) The hot cluster log is scanned, the hot clusters are read in, and their contents are used to initialize in-memory meta-data. (3) The file-to-cluster mappings are read in from disk. (4) The delete log is scanned and records are applied, modifying meta-data as necessary. (5) Proxy service is enabled. Hummingbird will now service requests. (6) During idle time, the recovery process scans the cluster region, rebuilding the in-memory meta-data for all files and clusters. As files are recovered, they are available to the application. (7) Recovery is complete. All files and clusters are now available. Hummingbird can now write new clusters to disk.

3.5 Daemons

Hummingbird employs a number of daemons in addition to the recovery daemons mentioned above that run when the file system is idle and can be called on demand. In the future, we hope to support client-provided daemons, which would, for example, support different cache eviction policies.

Pack files daemon. If the amount of main memory used to store files exceeds a tunable threshold, the `pack_files_daemon()` uses the file LRU list to create a cluster of files and write the cluster to disk. The daemon packs the files using the information from the `collocate_file()` calls, attempting to pack files from the same locality set in the same cluster. If a file is larger than the cluster size, it is split between multiple clusters.

This daemon uses the *disk data layout policy* to decide which cluster to pack next. Implemented policies include: *closest cluster*, which picks the closest free cluster to the previous cluster on disk accessed, *closest cluster to previous write*, which picks the closest free cluster to the previous cluster written to on disk, and *overwrite*, which overwrites the next cluster. All of these policies tend to write files accessed within a short time to clusters close together on disk. Thus, long-term fragmentation does not occur.

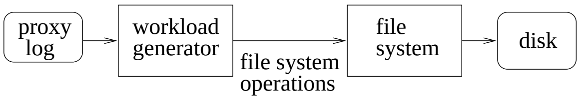

Figure 1: Simulation environment.

Pack files from head daemon. The `pack_files_daemon()` takes files off the tail of the LRU list to pack into clusters; `pack_files_from_head_daemon()` takes files off the head. This has the effect of packing the most-frequently accessed files into clusters.

Free main memory data daemon. This daemon evicts data using file and cluster LRU lists when the amount of main memory used by the file system exceeds a tunable threshold.

Free disk space daemon. This daemon deletes files or clusters whose age exceeds tunable thresholds set by the *delete file policy* and the *delete cluster policy*.

4 Experimental results

We built an environment to run trace-driven experiments on real implementations of UFS and Hummingbird. This environment consists of four components, as depicted in Figure 1. (1) The processed *proxy log* which was discussed in Section 2.1. (2) The *workload generator* simulates the operation of a caching web proxy. It reads the proxy log events and generates the file system operations that a proxy would have generated during processing of the original HTTP requests. (3) The *file system*, which is either Hummingbird or various implementations of UFS, EFS, or XFS. (4) The *disk*, which is a physical magnetic disk that the file system uses to store the files. The workload generators, implementation details, and experiments are described in the following section.

4.1 Workload generators

We developed two workload generators: `wg-Squid` which mimics Squid's interaction with the file system, and `wg-Hummingbird` which issues Hummingbird calls. The same wg-squid workload generator was used with the various UFS implementation, EFS and XFS. The generators take as input the modified proxy access log. The workload generators operate in a trace-driven loop which processes logged events sequentially without pause[2]. This simulates a heavily loaded server.

UFS workload generator: `wg-Squid`**.** The `wg-Squid` simulates the file system behavior of Squid-2.2. The Squid cache has a 3-level directory hierarchy for storing files; the number of children at each level is a configurable parameter. In order to minimize file lookup times, Squid attempts to keep directories small by distributing files evenly across the directory hierarchy.

When a file is written to the cache a top-level directory, or *SwapDir*, is selected. Squid attempts to load balance across the SwapDirs. Once the SwapDir has been selected, the file is assigned a file number which uniquely identifies the file within the SwapDir. The value of this file number is used to compute the names of the level-2 and level-3 directories. Thus, Squid does not use the URL or URL reference locality for file placement into directories, limiting the ability of the file system to collocate files which will be accessed together. Once the directory path has been determined, the file is created and written.

Squid has configurable high and low water marks. When the total cache size passes the high water mark, eviction begins. Files are deleted from the cache in modified LRU order with a small bias towards expired files. Eviction continues until the low water mark is reached. The `wg-Squid` simulates this behavior except that it uses LRU instead of modified LRU; our log does not contain the expires information.

Hummingbird workload generator: `wg-Hummingbird`**.** Each iteration of the `wg-Hummingbird` loop parses the next sequential event from the log and attempts to read the URL from Hummingbird. If successful, it explicitly releases the file buffer. If the read attempt fails, it attempts to write the URL into the file system; this would have occurred after the proxy fetched the URL from the server.

The `wg-Hummingbird` maintains a hash table keyed by client IP address to store information for generating the collocate hints. The values in the hash table are the URLs of the most recent HTML file request seen from a particular client. The hash table only stores the URLs of static HTML files. As requests for non-HTML documents are processed, `wg-Hummingbird` generates `collocate_files()` calls for the non-HTML paired with its client's current HTML file, as stored in the hash table.

Note that Squid (and `wg-Squid`) do not provide explicit locality hints to the underlying operating system; they only place files in the directory hierarchy. This is in contrast with `wg-Hummingbird`, which provides explicit locality hints via the `colocate_files()` call. There are other ways which a proxy could use UFS to obtain file locality; we did not test against these other approaches. Thus, our comparisons are restricted to the Squid approach for file locality.

4.2 Experiments

We performed our experiments on PCs with a 700 MHz Pentium III running FreeBSD 4.1 with an 18 GB IBM 10,000 RPM Ultra2 LVD SCSI (Ultrastar 18LZX). We also performed experiments on an SGI Origin 2000 with 18 GB 10,000 RPM Seagate Cheetah disks (ST118202FC) under IRIX 6.5. The results with different parameter settings were similar to each other so we only present a representative subset of the results.

Our experiments used the 4-day web proxy log as input into the workload generators. Among other measurements, we measured the *proxy hit rate*, which represents how frequently the web page will not have to be fetched from the server, and the *file system read time*, which represents how long the proxy must wait to get the file from the file system. The *file system write time* is the time it takes the file system to return after a write; this may not include the time to write the file or file metadata to disk. (We call the file system read (write) times *FS read (write) time* in our figures and tables.) `wg-Squid` and Hummingbird measured the file system read/write times directly, and the output of the `iostat -o` and `sar` commands were used to determine the disk I/O times. We also report the *throughput*, which we compute as the ratio of the experiment run time and the total number of requests processed. The measurements are averaged over the entire log; warm-up effects were insignificant.

Comparing Hummingbird with UFS: single thread. We compared Hummingbird with three versions of UFS on FreeBSD 4.1. The three versions of UFS were: UFS, which is UFS mounted

[2] We call this timing mode THROTTLE. We have implemented two other different timing modes: REAL and FAST. REAL issues the event with the frequency that they were recorded. FAST processes the events with a speed-up heuristic parameterized by n; if the time between events is longer than time n, then we only wait time n between them.

Table 2: Comparing Hummingbird with UFS, UFS-async, and UFS-soft when files greater than 64 KB are not cached.

file system	disk size	main memory (MB)	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)
Hummingbird	4 GB	256	0.62	1.81	0.32	1,161,163	5.14
UFS-async	4 GB	128+128	0.64	6.13	2.54	4,807,440	4.66
UFS-soft	4 GB	128+128	0.64	5.54	21.07	10,737,300	4.97
UFS	4 GB	128+128	0.64	5.93	20.77	10,238,460	5.02
Hummingbird	4 GB	1024	0.63	1.05	0.03	552,882	5.75
UFS-async	4 GB	512+512	0.64	3.21	2.35	3,217,440	3.95
UFS-soft	4 GB	512+512	0.64	3.27	20.85	9,714,420	4.76
UFS	4 GB	512+512	0.64	3.92	18.43	9,022,200	4.63
Hummingbird	8 GB	256	0.64	2.03	0.32	1,194,494	5.64
UFS-async	8 GB	128+128	0.67	5.69	1.47	4,605,360	4.34
UFS-soft	8 GB	128+128	0.67	5.78	15.66	9,058,141	4.86
UFS	8 GB	128+128	0.67	6.17	12.83	8,313,360	4.63
Hummingbird	8 GB	1024	0.64	1.20	0.03	578,562	6.37
UFS-async	8 GB	512+512	0.67	4.05	1.34	3,561,180	4.13
UFS-soft	8 GB	512+512	0.67	3.74	15.65	8,148,721	4.62
UFS	8 GB	512+512	0.67	3.81	15.70	7,611,840	4.68

synchronously (the default), UFS-soft, which is UFS with soft updates, and UFS-async, which is UFS mounted asynchronously, so that meta-data updates are not synchronous and the file system is not guaranteed to be recoverable after a crash. We used a version of Hummingbird with a single working thread, where the daemons were called explicitly every 1000 log events. Table 2 presents comparisons for two different disk sizes, 4 GB and 8 GB, with two memory sizes, 256 MB and 1024 MB, when files greater than 64 KB are not cached. The memory was split evenly between the Squid cache and the file system buffer cache[3]. The proxy-perceived latency in Table 2 is the 5th column, the FS read time. Hummingbird's smaller file system read time is due to the hits in main memory caused by grouping files in locality sets into clusters. Hummingbird's smaller file system write time (6th column) when compared to UFS-async is due to cluster writes, which write multiple files to disk in a single operation. The FS write times for UFS and UFS-soft are greater than UFS-async due to the synchronous file create operation.

The effectiveness of the clustered reads and writes and the collocation strategy is illustrated in the number of disk I/Os. In all test configurations, Hummingbird issued substantially fewer disk I/Os than any of the UFS configurations. Also, note that the number of disk I/Os in the UFS experiments is larger than the total number of requests in the log. This is because file operations resulted in multiple disk I/Os. This also explains why UFS read and write operations (as seen in FS read and write times) are slower than individual disk I/Os. The mean disk I/O time is larger in Hummingbird since the request size is a cluster, which is larger when compared to the mean data transfer size accessed by UFS.

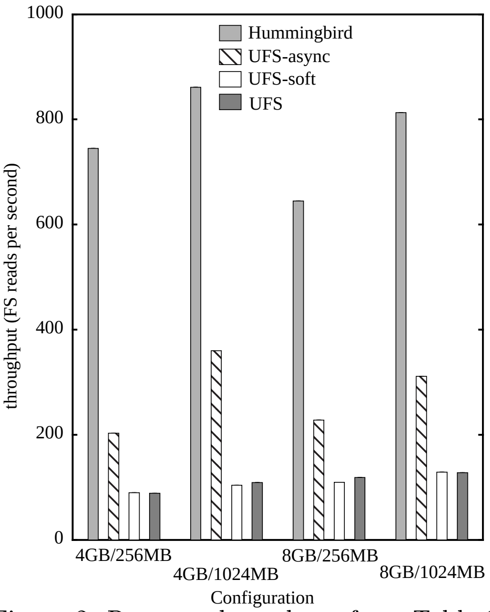

Figure 2: Request throughput from Table 2.

The throughput for each experiment in Table 2 is

[3]We controlled the size of the file system buffer cache by locking the Squid's memory using the `mlock` system call and locking additional memory, so that file system buffer cache could use only the remaining unlocked memory. In this way we also prevented paging of the Squid process.

shown in Figure 2. Figure 2 shows that Hummingbird throughput is much higher than both UFS, UFS-soft, and UFS-async on the same disk size and memory size. This is not quite a fair comparison since the proxy hit rate is lower with Hummingbird. (We do not expect the experiment run time to increase more than 10% when the Hummingbird policies are set so that it would have equivalent hit rate to `wg-Squid`). The throughput is larger for Hummingbird since much less time is spent in disk I/O. Using throughput as a comparison metric, we see that Hummingbird is 2.3–4.0 times faster than simulated Squid running on UFS-async, 5.6–8.4 times larger than a simulated version of Squid running on UFS-soft, and 5.4–9.4 times faster than simulated Squid running on UFS. These numbers include also the results from Table 3.

The experiments for Table 2 assumed that files greater than 64 KB were not cached by the proxy. We got similar results when assuming the proxy would cache all files; see Table 3. Note that the proxy hit rate in Table 3 is lower than in Table 2. This is the result of the cache being "polluted" with large files, which cause some smaller files to be evicted. The end result is that there are less hits, which translate into less file system activity, and fewer file accesses.

Comparing Hummingbird with EFS and XFS: single thread. We compared Hummingbird with EFS and XFS on SGI IRIX. EFS is an extent file system. XFS is a high-performance journalling file system that interacts with the kernel through the traditional VFS and vnode interfaces.

We experimented with 3 different disk sizes, 4 GB, 9 GB, and 18 GB. Since EFS supports file systems up to 8 GB in size, we could not test it with 9 or 18 GB disks. Table 4 presents results where `wg-Squid` cache and the IRIX buffer cache together use 256 MB of main memory, which are divided in two ways: 50 MB + 206 MB and 128 MB + 128 MB (`wg-Squid` and buffer cache, respectively). The 50 MB for the `wg-Squid` was selected according to the Squid administration guidelines [23]. Hummingbird has 256 MB of main memory, and it has the best choice of policies as previously discussed.

Our experimental results are in Table 4. Hummingbird throughput is much higher than both XFS and EFS on the same disk size as seen in the UFS experiments, and the user-perceived latency (5th column in Table 4) is smallest in Hummingbird. The second observation from Table 4 is that EFS is much slower than XFS, which is a journalling file system. In particular, file system write time for EFS is more than 3 times larger than XFS write time. It is the result of frequent synchronous write operations for meta-data, which are performed by EFS. The third observation is that increasing `wg-Squid` cache size to 128MB and reducing the file system buffer cache size actually improved the file system performance, which indicates that the `wg-Squid` cache is more effective than the file system buffer cache.

Comparing Hummingbird with UFS: multi-threaded workload generators. Many proxies are either multi-threaded or have multiple processes. Hummingbird is both thread- and process-safe. We implemented multi-threaded versions of both our workload generators, with a thread for the daemons in `wg-Hummingbird`, and ran similar experiments to the above with one processor running FreeBSD 4.1 with the LinuxThreads library. Table 5 contains a subset of the results of our experiments using four threads and two disks. In multi-threaded squid workload generator, two cache root directories are used, each residing on a disk. The experiment run time for the experiments in Table 5 are consistently longer than the corresponding cases in Table 3 due to an uneven distribution of the files on the 2 disks; we observed a bursty access pattern to each disk in the `iostat` log. Queuing in the device driver results in longer FS read/write time too. However, the Hummingbird throughput is again 2–4 times greater than for UFS, UFS-async, and UFS-soft.

Recovery performance. Recovering the hot clusters and the deletion log are quick; it takes about 30 seconds to read 2500 hot clusters and a log of 3000 deletion entries into memory. Thus, on a system crash with a 18 GB disk, Hummingbird can start to service requests in 30 seconds, while UFS will take more than 20 minutes to perform the necessary `fsck` before requests can be serviced. While rebuilding the in-memory meta-data (Step 6 in Section 3.4), the FS read time increased a small amount (e.g., from 2.03 ms to 2.50 ms for a 8 GB disk).

Journalling file systems will recover integrity quickly compared with traditional UFSs due to log-based recovery. However, journalling alone will not fetch hot data from disk as part of the recovery process.

5 Polygraph results

A web cache benchmarking package called Web Polygraph [18] is used for comparison of caching

Table 3: Comparing Hummingbird with UFS, UFS-async, and UFS-soft when all files are cached.

file system	disk size	main memory (MB)	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)	experiment run time (s)
Hummingbird	4 GB	256	0.60	1.68	0.39	1,349,175	4.17	6,362
UFS-async	4 GB	128+128	0.62	6.61	2.88	5,134,380	4.72	25,510
UFS-soft	4 GB	128+128	0.62	5.81	22.91	11,858,100	5.02	59,475
UFS	4 GB	128+128	0.62	6.13	22.67	11,283,060	5.05	59,807
Hummingbird	4 GB	1024	0.61	0.88	0.52	630,362	5.62	6,030
UFS-async	4 GB	512+512	0.62	3.67	2.70	3,919,620	3.98	16,384
UFS-soft	4 GB	512+512	0.62	3.51	22.89	10,868,700	4.84	52,377
UFS	4 GB	512+512	0.62	4.24	20.23	10,115,400	4.69	49,749
Hummingbird	8 GB	256	0.64	2.17	0.40	1,464,727	5.03	7,923
UFS-async	8 GB	128+128	0.66	6.42	2.30	5,017,920	4.67	24,657
UFS-soft	8 GB	128+128	0.66	6.14	19.31	10,461,841	4.98	51,797
UFS	8 GB	128+128	0.66	7.37	16.42	9,954,540	4.89	51,062
Hummingbird	8 GB	1024	0.62	1.18	0.51	695,771	6.41	6,802
UFS-async	8 GB	512+512	0.66	4.66	1.90	4,016,400	4.32	18,240
UFS-soft	8 GB	512+512	0.67	3.69	15.71	8,158,080	4.61	37,097
UFS	8 GB	512+512	0.66	4.24	18.90	8,894,760	4.82	45,044

Table 4: Comparing Hummingbird with XFS and EFS with 256 MB of main memory when all files are cached.

file system	disk size	cache size (MB)	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)	experiment run time (s)
Hum	4 GB	256	0.62	2.82	0.20	922,871	9.85	9,926
EFS	4 GB	50+206	0.62	14.79	46.38	10,784,969	10.99	128,878
XFS	4 GB	50+206	0.62	14.97	16.56	10,870,353	6.67	75,565
EFS	4 GB	128+128	0.62	14.08	48.33	10,540,778	11.37	130,359
XFS	4 GB	128+128	0.62	14.90	13.95	10,115,369	6.72	70,746
Hum	9 GB	256	0.66	3.30	0.21	1,028,922	10.77	11,861
XFS	9 GB	50+206	0.66	15.47	12.39	9,558,954	7.02	69,929
XFS	9 GB	128+128	0.66	15.65	8.80	8,737,878	7.12	64,726
Hum	15 GB	256	0.67	3.71	0.21	1,049,121	11.92	13,313
XFS	15 GB	50+206	0.67	16.71	5.73	8,103,576	7.60	63,371
XFS	15 GB	128+128	0.67	15.91	5.79	7,841,184	7.55	60,943

Table 5: Comparing Hummingbird with UFS, UFS-async, and UFS-soft with 256 MB of main memory when all files are cached with 2 disks and 4 threads in the workload generator.

file system	disk size	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)	experiment run time (s)
Hummingbird	4 GB	0.64	4.86	1.34	1,478,005	11.68	11,743
UFS-async	4 GB	0.66	5.92	6.39	5,638,201	10.72	30,427
UFS-soft	4 GB	0.66	3.25	20.42	9,405,301	9.76	45,655
UFS	4 GB	0.66	6.71	15.86	10,771,201	9.05	48,577
Hummingbird	8 GB	0.67	6.12	1.40	1,556,169	14.08	12,997
UFS-async	8 GB	0.67	6.19	5.18	5,466,061	10.67	29,354
UFS-soft	8 GB	0.67	6.04	14.67	8,678,761	10.52	44,622
UFS	8 GB	0.67	6.78	9.73	8,903,641	8.74	38,464

web proxies. The clients and servers are simulated; the client workload parameters such as hit ratio, cacheability, and response sizes can be specified and server-side delays can be specified. We used the PolyMix-2 traffic model to compare Apache [25] using UFS and a slightly modified version of Apache using Hummingbird without `collocate_files()` calls. Due to space considerations of this paper, we only briefly discuss our results.

We used one of our FreeBSD Pentium IIIs for the proxy. We used a number of different client request rates; the following results are for 8 requests/second. We found that the mean response time for hits in the proxy is 14 times smaller with Hummingbird than with UFS, and the median response time for hits is 20 times smaller. The improvement in mean and median response times for proxy misses was much smaller, as expected; Hummingbird is 20% faster than UFS. Since Hummingbird serves proxy hits faster, its request queue is shorter, which in turn, shortens the queue time of proxy misses.

6 Related work

Related work falls into two categories: first, analyses of traditional UFS-based systems and ways to beat their performance limitations, and second, analyses of the behavior of web proxies and how they can better use the underlying I/O and file systems.

The first set of research extends back to the original FFS work of McKusick et al. [13] which addressed the limitations of the System V file system by introducing larger block sizes, fragments, and cylinder groups. With increasing memory and buffer cache sizes, UNIX file systems were able to satisfy more reads out of memory. The FFS clustering work of McVoy and Kleiman [14] sought to improve write times by lazily writing the data to disk in contiguous extents called *clusters*. LFS [20] sought to improve write times by packing dirty file blocks together and writing to an on-disk log in large extents called *segments*. The LFS approach necessitates a cleaner daemon to coalesce live data and free on-disk segments. As well, new on-disk structures are required. Work in soft updates [6] and journalling [7, 4] has sought to alleviate the performance limitations due to synchronous meta-data operations, such as file create or delete, which must modify file system structures in a specified order. Soft updates maintains dependency information in kernel memory to order disk updates. Journalling systems write meta-data updates to an auxiliary log using the write-ahead logging protocol. This differs from LFS, in which the log contains all data, including meta-data. LFS also addresses the meta-data update problem by ordering updates within segments.

The Bullet server [26, 24] is the file system for Amoeba, a distributed operating system. The Bullet service supports entire file operations to read, create, and delete files. All files are immutable. Each file is stored contiguously, both on disk and in memory. Even though the file system API is similar to Hummingbird, the Bullet service does not perform clustering of files together, so it would not have the same type of performance improvement that Hummingbird has for a caching web proxy workload.

Kaashoek et al. [9] approaches high performance through developing *server operating systems*, where a server operating system is a set of abstractions and runtime support for specialized, high performance server applications. Their implementation of the Cheetah web server is similar to Hummingbird in one way: collocating an HTML page and its images on disk and reading them from disk as a unit. Web servers' data storage is represented naturally by the UFS file hierarchy. This is not true for caching web proxies as discussed in Section 2.2.

CacheFlow [2] builds a cache operating system called CacheOS with an optimized object storage system which minimizes the number of disk seek and I/O operations per object. Unfortunately, details of the object storage are not public. The Network Appliance filer [8] is a prime example of combination of an operating system and a specialized file system (WAFL) inside a storage appliance. Novell [15] has developed the Cache Object Store (COS) which they state is 10 times more efficient than typical file systems; few details on the design are available. The COS prefetches the components for a page when the page is requested, leading us to believe that the components are not stored contiguously as they are in Hummingbird.

Rousskov and Soloviev [21] studied the performance of Squid and its use of the file system. Markatos et al. [12] presents methods for web proxies to work around costly file system file opens, closes, and deletes. One of their methods, LAZY-READS, gathers read requests n-at-a-time, and issues them all at the same time to the disk; results are presented when n is 10. This is similar to our clustering of locality sets, since a read for a cluster will, on average, access 8 files. We feel that Hummingbird in a more general solution to the decreasing the effect of costly file system operations on a web proxy.

Maltzahn et al. [10] compared the disk I/O of Apache and Squid and concluded that they were remarkably similar. In a later paper [11], they simulated the operation of several techniques for modifying Squid, one of which was to use a memory-mapped interface to access small files. Other techniques improved the locality of related files based on domain names. This paper reported a reduction of up to 70% in the number of disk operations relative to unmodified Squid. An inherent problem using one memory-mapped file to access all small objects is that it cannot scale to handle a very large number of objects. Like Hummingbird, using memory-mapped files requires modification to the proxy code.

Pai et al. [17] developed a kernel I/O system called IO-lite to permit sharing of "buffer aggregates" between multiple applications and kernel subsystems. This system solves the multiple buffering problem, but, like Hummingbird, applications must use a different interface that supersedes the traditional UNIX read and writes.

7 Conclusions

This paper explores file system support for applications which can take advantage of the performance/persistence tradeoff. Such a file system is especially useful for local caching of data, where permanent storage of the data is available elsewhere. A caching web proxy is the prime example of an application that may benefit from this file system.

We implemented Hummingbird, a light-weight file system that is designed to support caching web proxies. Hummingbird has two distinguishing features: it stores its meta-data in memory, and it stores groups of related objects (e.g., HTML page and its embedded images) together on the disk. By setting the tunable parameters to achieve persistence, Hummingbird can also be used to improve the performance of web servers, which have similar reference locality as proxies. Our results are very promising; Hummingbird's throughput is 2.3–4.0 times larger than a simulated version of Squid running UFS mounted asynchronously on FreeBSD, 5.4–9.4 times larger than a simulated version of Squid running UFS mounted synchronously on FreeBSD, 5.6–8.4 times larger than a simulated version of Squid running UFS with soft updates on FreeBSD, and 5.4–13 times larger than XFS and EFS on IRIX. The Web Polygraph environment confirmed these improvements for the response times for proxy hits.

Additional information about Hummingbird is available at `http://www.bell-labs.com/project/hummingbird/`.

Acknowledgements. Many thanks to Arthur Goldberg for giving us copies of the ISP log files that we studied. WeeTeck Ng helped us with preparing our experiment environment. Bruce Hillyer and Philip Bohannon were very helpful in initial design discussions. Hao Sun performed the Polygraph experiments.

References

[1] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar (Pacific Grove), CA, Oct. 1991), ACM Press, pp. 198–212.

[2] Network cache performance measurements. CacheFlow White Papers Version 2.1, CacheFlow, Sept. 1998.

[3] Cao, P., and Irani, S. Cost-aware WWW proxy caching algorithms. 193–206.

[4] Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., and Sidebotham, R. N. The Episode file system. In *Proceedings of the Winter 1992 USENIX Conference* (San Francisco, CA, Winter 1992), pp. 43–60.

[5] Gabber, E., and Shriver, E. Let's put NetApp and CacheFlow out of business! In *Proceedings of the 9th ACM SIGOPS European Workshop* (Kolding, Denmark, Sept. 2000), pp. 85–90.

[6] Ganger, G. R., and Patt, Y. N. Metadata update performance in file systems. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, Nov. 1994), pp. 49–60.

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

[8] Hitz, D., Lau, J., and Malcolm, M. File system design for an NFS file server appliance. In

Proceedings of the USENIX 1994 Winter Technical Conference (San Francisco, CA, Jan. 1994), pp. 235–246.

[9] Kaashoek, M. F., Engler, D. R., Ganger, G. R., and Wallach, D. A. Server operating systems. In *Proceedings of the 1996 SIGOPS European Workshop* (Connemara, Ireland, Sept. 1996), pp. 141–148.

[10] Maltzahn, C., Richardson, K., and Grunwald, D. Performance issues of enterprise level proxies. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)* (Seattle, WA, June 1997), pp. 13–23.

[11] Maltzahn, C., Richardson, K. J., and Grunwald, D. Reducing the disk I/O of web proxy server caches. In *Proceedings of the 1999 USENIX Annual Technical Conference* (Monterey, CA, June 1999), pp. 225–238.

[12] Markatos, E. P., Katevenis, M. G., Pnevmatikatos, D., and Flouris, M. Secondary storage management for web proxies. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems* (Boulder, CO, Oct. 1999), pp. 93–114.

[13] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 181–197.

[14] McVoy, L. W., and Kleiman, S. R. Extent-like performance from a UNIX file system. In *Proceedings of the Winter 1991 USENIX Conference* (Dallas, TX, Jan. 1991), pp. 33–43.

[15] The Novell ICS advantage: Competitive white paper. Tech. rep. Available at `http://www.novell.com/advantage/nics/nics-compwp.html`.

[16] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, Dec. 1985), vol. 19, ACM Press, pp. 15–24.

[17] Pai, V. S., Druschel, P., and Zwaenepoel, W. IO-lite: a unified I/O buffering and caching system. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI'99)* (New Orleans, LA, Feb. 1999), pp. 15–28.

[18] Polyteam. Web polygraph site. `http://polygraph.ircache.net/`.

[19] Roselli, D., Lorch, J. R., and Anderson, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)* (San Diego, CA, June 2000), pp. 41–54.

[20] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26–52.

[21] Rousskov, A., and Soloviev, V. A performance study of the Squid proxy on HTTP/1.0. *World-Wide Web Journal, Special Edition on WWW Characterization and Performance and Evaluation 2*, 1–2 (1999).

[22] Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., and Stein, C. A. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)* (San Diego, CA, June 2000), pp. 71–84.

[23] 1999. `http://squid.nlanr.net/mail-archive/squid-users/`.

[24] Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, J., and van Rossum, G. Experiences with the Amoeba distributed operating system. *Communications of the ACM 33* (Dec. 1990), 46–63.

[25] The Apache Software Foundation. Apache http server project. `http://www.apache.org/httpd.html`.

[26] van Renesse, R., Tanenbaum, A. S., and Wilschut, A. N. The design of a high-performance file server. In *Proceedings of the Ninth International Conference on Distributed Computing Systems (ICDCS 1989)* (Newport Beach, CA, June 1989), IEEE Computer Society Press, pp. 22–27.