Reducing the Disk I/O of Web Proxy Server Caches

Carlos Maltzahn and Kathy J. Richardson
Compaq Computer Corporation
Network Systems Laboratory
Palo Alto, CA
carlosm@cs.colorado.edu, kjr@pa.dec.com

Dirk Grunwald
University of Colorado
Department of Computer Science
Boulder, CO
grunwald@cs.colorado.edu

Abstract

The dramatic increase of HTTP traffic on the Internet has resulted in wide-spread use of large caching proxy servers as critical Internet infrastructure components. With continued growth the demand for larger caches and higher performance proxies grows as well. The common bottleneck of large caching proxy servers is disk I/O. In this paper we evaluate ways to reduce the amount of required disk I/O. First we compare the file system interactions of two existing web proxy servers, CERN and SQUID. Then we show how design adjustments to the current SQUID cache architecture can dramatically reduce disk I/O. Our findings suggest two that strategies can significantly reduce disk I/O: (1) preserve locality of the HTTP reference stream while translating these references into cache references, and (2) use virtual memory instead of the file system for objects smaller than the system page size. The evaluated techniques reduced disk I/O by 50% to 70%.

1 Introduction

The dramatic increase of HTTP traffic on the Internet in the last years has lead to the wide use of large, enterprise-level World-Wide Web proxy servers. The three main purposes of these web proxy servers are to control and filter traffic between a corporate network and the Internet, to reduce user-perceived latency when loading objects from the Internet, and to reduce bandwidth between the corporate network and the Internet. The latter two are commonly accomplished by caching objects on local disks.

Apart from network latencies the bottleneck of Web cache performance is disk I/O [2, 27]. An easy but expensive solution would be to just keep the entire cache in primary memory. However, various studies have shown that the Web cache hit rate grows in a logarithmic-like fashion with the amount of traffic and the size of the client population [16, 11, 8] as well as logarithmic-proportional to the cache size [3, 15, 8, 31, 16, 25, 9, 11] (see [6] for a summary and possible explanation). In practice this results in cache sizes in the order of ten to hundred gigabytes or more [29]. To install a server with this much primary memory is in many cases still not feasible.

Until main memory becomes cheap enough, Web caches will use disks, so there is a strong interest in reducing the overhead of disk I/O. Some commercial Web proxy servers come with hardware and a special operating system that is optimized for disk I/O. However, these solutions are expensive and in many cases not affordable. There is a wide interest in portable, low-cost solutions which require not more than standard off-the-shelf hardware and software. In this paper we are interested in exploring ways to reduce disk I/O by changing the way a Web proxy server application utilizes a general-purpose Unix file system using standard Unix system calls.

In this paper we compare the file system interactions of two existing web proxy servers, CERN [18] and SQUID [30]. We show how adjustments to the current

SQUID cache architecture can dramatically reduce disk I/O. Our findings suggest that two strategies can significantly reduce disk I/O: (1) preserve locality of the HTTP reference stream while translating these references into cache references, and (2) use virtual memory instead of the file system for objects smaller than the system page size. We support our claims using measurements from actual file systems exercised by a trace driven workload collected from proxy server log data at a major corporate Internet gateway.

In the next section we describe the cache architectures of two widely used web proxy servers and their interaction with the underlying file systems. We then propose a number of alternative cache architectures. While these cache architectures assume infinite caches we investigate finite cache management strategies in section 3 focusing on disk I/O. In section 4 we present the methodology we used to evaluate the cache structures and section 5 presents the results of our performance study. After discussing related work in section 6, we conclude with a summary and future work in section 7.

2 Cache Architectures of Web Proxy Servers

We define the *cache architecture* of a Web proxy server as the way a proxy server interacts with a file system. A cache architecture names, stores, and retrieves objects from a file system, and maintains application-level meta-data about cached objects. To better understand the impact of cache architectures on file systems we first review the basic design goals of file systems and then describe the Unix Fast File System (FFS), the standard file system available on most variants of the UNIX operating system.

2.1 File systems

Since the speed of disks lags far behind the speed of main memory the most important factor in I/O performance is *whether* disk I/O occurs at all ([17], page 542). File systems use memory caches to reduce disk I/O. The file system provides a *buffer cache* and a *name cache*. The buffer cache serves as a place to transfer and cache data to and from the disk. The name cache stores file and directory *name resolutions* which associate file and directory names with file system data structures that otherwise reside on disk.

The Fast File System (FFS) [20] divides disk space into *blocks* of uniform size (either 4K or 8K Bytes). These are the basic units of disk space allocation. These blocks may be sub-divided into *fragments* of 1K Bytes for small files or files that require a non-integral number of blocks. Blocks are grouped into *cylinder groups* which are sets of typically sixteen adjacent cylinders. These cylinder groups are used to map file reference locality to physically adjacent disk space. FFS tries to store each directory and its content within one cylinder group and each file into a set of adjacent blocks. The FFS does not guarantee such file layout but uses a simple set of heuristics to achieve it. As the file system fills up, the FFS will increasingly often fail to maintain such a layout and the file system gets increasingly *fragmented*. A fragmented file system stores a large part of its files in non-adjacent blocks. Reading and writing data from and to non-adjacent blocks causes longer seek times and can severely reduce file system throughput.

Each file is described by meta-data in the form of *inodes*. An inode is a fixed length structure that contains information about the size and location of the file as well as up to fifteen pointers to the blocks which store the data of the file. The first 12 pointers are direct pointers while the last three pointers refer to *indirect blocks*, which contain pointers to additional file blocks or to additional indirect blocks. The vast majority of files are shorter than 96K Bytes, so in most cases an inode can directly point to all blocks of a file, and storing them within the same cylinder group further exploits this reference locality.

The design of the FFS reflects assumption about file system workloads. These assumptions are based on studies of workloads generated by workstations [23, 24]. These workstation workloads and Web cache request workloads share many, but not all of the same characteristics. Because most of their behavior is similar, the file system works reasonably well for caching Web pages. However there are differences; and tweaks to the way cache objects map onto the file system produce significant performance improvements.

We will show in the following sections that some file system aspects of the workload characteristics generated by certain cache architectures can differ from usual workstation workloads. These different workload characteristics lead to poor file system performance. We will also show that adjustments to cache architectures can dramatically improve file system performance.

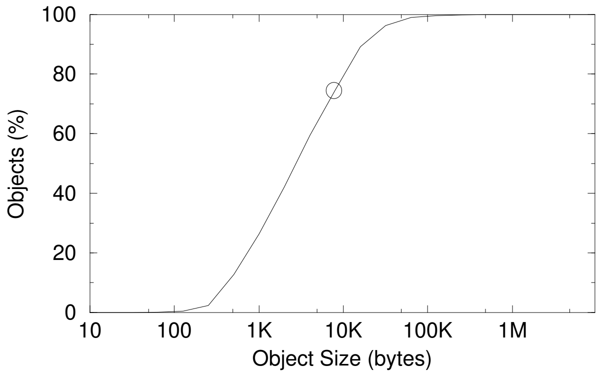


Figure 1: The dynamic size distribution of cached objects. The graph shows a cumulative distribution weighted by the number of objects. For example 74% of all object referenced have a size of equal or less than 8K Bytes.

2.2 File System Aspects of Web Proxy Server Cache Workloads

The basic function of a Web proxy server is to receive a request from a client, check whether the request is authorized, and serve the requested object either from a local disk or from the Internet. Generally, objects served from the Internet are also stored on a local disk so that future requests to the same object can be served locally. This functionality combined with Web traffic characteristics implies the following aspects of Web proxy server cache generated file system loads:

Entire Files Only Web objects are always written or read in their entirety. Web objects do change, but this causes the whole object to be rewritten; there are no incremental updates of cached Web objects. This is not significantly different than standard file system workloads where more than 65% of file accesses either read or write the whole file. Over 90% either read or write sequentially a portion of a file or the whole file [5]. Since there are no incremental additions to cached objects, it is likely that disk becomes more fragmented since there are fewer incremental bits to utilize small contiguous block segments.

Size Due to the characteristics of Web traffic, 74% of referenced Web objects are smaller than 8K Bytes. Figure 1 illustrates this by showing the distribution of the sizes of cached objects based on our HTTP traces, which are described later. This distribution is very similar to file characteristics. 8K Byte is a common system page size. Modern hardware supports the efficient transfer of system page sizes between disk and memory. A number of Unix File Systems use a file system block size of 8K Bytes and a fragment size of 1K Bytes. Typically the performance of the file system's fragment allocation mechanism has a greater impact on overall performance than the block allocation mechanism. In addition, fragment allocation is often more expensive than block allocation because fragment allocation usually involves a best-fit search.

Popularity The popularity of Web objects follows a *Zipf-like* distribution Ω/i^α (where $\Omega = (\sum_{i=1}^{N} 1/i^\alpha)^{-1}$ and i is the ith most popular Web object) [15, 6]. The α values range from 0.64 to 0.83. Traces with homogeneous communities have a larger α value than traces with more diverse communities. The traces generally do not follow Zipf's law which states that $\alpha = 1$ [33]. The relative popularity of objects changes slowly (on the order of days and weeks). This implies that for any given trace of Web traffic, the first references to popular objects within a trace tend to occur early in the trace. The slow migration to new popular items allows for relatively static working set capture algorithms (see for example [28]). It also means that there is little or no working set behavior attributable to the majority of the referenced objects. File system references exhibit much more temporal locality; allocation and replacement policies need to react rapidly to working set changes.

Data Locality A large number of Web objects include links to embedded objects that are referenced in short succession. These references commonly refer to the same server and tend to have the same URL prefix. This is similar to the locality observed in workstation workloads which show that files accessed in short succession tend to be in the same file directory.

Meta-data Locality The fact that objects with similar names tend to be accessed in short succession means that information about those objects will also be referenced in short succession. If the information required to validate and access files is combined in the same manner as the file accesses it will exhibit temporal locality (many re-references within a short time period). The hierarchal directory structure of files systems tends to group related files together. The meta-data about those files and their access methods are stored in directory and inodes which end up being highly reused when accessing a group of files. Care is required to properly map Web objects to preserve the locality of meta-data.

Read/Write Ratio The hit rate of Web caches is low (30%-50%, see [15, 1, 31, 4]). Every cache hit involves a read of the cache meta-data and a read of the cached data. Every miss involves a read of the cache meta-data, a write of meta-data, and a write of the Web object. Since there are typically more misses than hits, the majority of disk accesses are writes. File systems typically have many more reads than writes [23]; writes require additional work because the file system data must be properly flushed from memory to disk. The high fraction of writes also causes the disk to quickly fragment. Data is written, removed and rewritten quickly; this makes it difficult to keep contiguous disk blocks available for fast or large data writes.

Redundancy Cached Web objects are (and should be) redundant; individual data items are not critical for the operation of Web proxy caches. If the cached data is lost, it can always be served from the Internet. This is not the case with file system data. Data lost before it is securely written to disk is irrecoverable. With highly reliable Web proxy servers (both software and hardware) it is acceptable to never actually store Web objects to disk, or to periodically store all Web objects to disk in the event of a server crash. This can significantly reduce the memory system page replacement cost for Web objects. A different assessment has to be made for the meta-data which some web proxy server use for cache management. In the event of meta-data loss, either the entire content of the cache is lost or has to be somehow rebuilt based on data saved on disk. High accessibility requirements might neither allow the loss of the entire cache nor time consuming cache rebuilds. In that case meta-data has to be handled similarly to file system data. The volume of meta-data is however much smaller than the volume of cached data.

2.3 Cache Architectures of Existing Web Proxy Servers

The following describes the cache architectures we are investigating in this paper. The first two describe the architectures of two widely used web proxy servers, CERN and SQUID. We then describe how the SQUID architecture could be changed to improve performance. All architectures assume infinite cache sizes. We discuss the management of finite caches in section 3.

2.3.1 CERN

The original web server `httpd` was developed at CERN and served as early reference implementation for World-Wide Web service. `httpd` can also be used as a web proxy server [18]. We refer to this function of httpd as "CERN".

CERN forks a new process for each request and terminates it after the request is served. The forked processes of CERN use the file system not only to store cached copies of Web objects but also to share meta-information about the content of the cache and to coordinate access to the cache. To find out whether a request can be served from the cache, CERN first translates the URL of the request into a URL directory and checks whether a *lock file* for the requested URL exists. The path of the URL directory is the result of mapping URL components to directories such that the length of the file path depends on the number of URL components. The check for a lock file requires the translation of each path component of the URL directory into an inode. Each translation can cause a miss in the file system's name cache in which case the translation requires information from the disk.

The existence of a lock file indicates that another CERN process is currently inserting the requested object into the cache. Locked objects are not served from the cache but fetched from the Internet without updating the cache. If no lock file exists, CERN tries to open a meta-data file in the URL directory. A failure to do so indicates a cache miss in which case CERN fetches the object from the Internet and inserts it into the cache thereby creating the necessary directories, temporary lock files, and meta-data file updates. All these operations require additional disk I/O in the case of misses in the file system's name and buffer cache. If the meta-data file exists and it lists the object file name as not expired, CERN serves the request from the cache.

2.3.2 SQUID

The SQUID proxy [30] uses a single process to eliminate CERN's overhead of process creation and termination. The process keeps meta-data about the cache contents in main memory. Each entry of the the meta-data maps a URL to a *unique file number* and contains data about the "freshness" of the cached object. If the meta-data does not contain an entry for the requested URL or the entry indicates that the cached copy is stale, the object is fetched from the Internet and inserted into the cache. Thus, with in-memory meta-data the disk is never touched to find out whether a request is a Web cache miss or a Web cache hit.

A unique file number n maps to a two-level file path that contains the cached object. The file path follows from the unique file number using the formula

$$(x, y, z) = (n \bmod l_1, n/l_1 \bmod l_2, n)$$

where (x, y, z) maps to the file path "x/y/z", and l_1 and l_2 are the numbers of first and second level directories. Unique file numbers for new objects are generated by either incrementing a global variable or reusing numbers from expired objects. This naming scheme ensures that the resulting directory tree is balanced. The number of first and second level directories are configurable to ensure that directories do not become too large. If directory objects exceed the size of a file block, directory look-up times increase.

2.4 Variations on the SQUID Cache Architecture

The main difference between CERN and SQUID is that CERN stores all state on disk while SQUID keeps a representation of the content of its cache (the meta-data) in main memory. It would seem straightforward to assume that CERN's architecture causes more disk I/O than SQUID's architecture. However, as we showed in [19], CERN's and SQUID's disk I/O are surprisingly similar for the same workload.

Our conjecture was that this is due to the fact that CERN's cache architecture preserves some of the locality of the HTTP reference stream, while SQUID's unique numbering scheme destroys locality. Although the CERN cache has a high file system overhead, the preservation of the spatial locality seen in the HTTP reference stream leads to a disk I/O performance comparable to the SQUID cache.

We have designed two alternative cache architectures for the SQUID cache that improve reference locality. We also investigated the benefits of circumventing the common file system abstractions for storing and retrieving objects by implementing *memory-mapped* caches. Memory mapped caches can reduce the number of file-system calls and effectively use large primary memories. However, memory-mapped caches also introduce more complexity for placement and replacement policies. We will examine several such allocation policies.

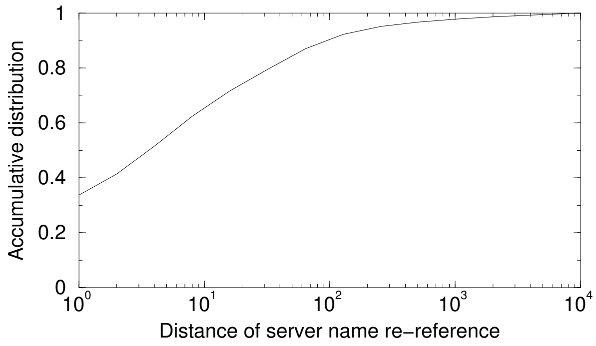


Figure 2: The locality of *server names* in an HTTP request stream. The data is based on an HTTP request stream with 495,662 requests (minus the first 100,000 to warm up the cache).

2.4.1 SQUIDL

We designed a modified SQUID cache architecture, SQUIDL, to determine whether a locality-preserving translation of an HTTP reference stream into a file system access stream reduces disk I/O. The only difference between SQUID and SQUIDL is that SQUIDL derives the URL directory name from the URL's host name instead of calculating a unique number. The modified formula for the file path of a cached object is now

$$(x, y, z) = (h(s) \wedge m_{l_1}, h(s) \wedge m_{l_2}, n)$$

where s is the host name of the requested URL, h is a hash function, $\wedge$ is the bitwise conjunction, m_{l_1} a bit mask for the first level directories, and m_{l_2} for the second level directories.

The rationale of this design is based on observation of the data shown in figure 2 (based on our HTTP traces, see below): the temporal locality of server names in HTTP references is high. One explanation for this is the fact that a large portion of HTTP requests are for objects that are embedded in the rendition of a requested HTML object. HTTP clients request these "in-lined" objects immediately after they parsed the HTML object. In most HTML objects all in-lined objects are from the same server. Since SQUIDL stores cached objects of the same server in the same directory, cache references to linked objects will tend to access the same directory. This leads to a burst of requests to the same directory and therefore increases the temporal locality of file system requests.

One drawback of SQUIDL is that a single directory may store many objects from a popular server. This can lead to directories with many entries which results in directory objects spanning multiple data blocks. Directory lookups in directory objects that are larger than one block can take significantly longer than directory lookups in single block directory objects [21]. If the disk cache is distributed across multiple file systems, directories of popular servers can put some file systems under a significantly higher workload than others. The SQUIDL architecture does produce a few directories with many files; for our workload only about 30 directories contained more than 1000 files. Although this skewed access pattern was not a problem for our system configuration, recent changes to SQUID version 2.0 [10, 13] implements a strategy that may be useful for large configurations. The changes balance file system load and size by allocating at most k files to a given directory. Once a directory reaches this user-configured number of files, SQUID switches to a different directory. The indexing function for this strategy can be expressed by

$$(x, y, z) = (n/(k * l_2), n/k \bmod l_2, n \bmod k)$$

where k is specified by the cache administrator. Notice that this formula poses an upper limit of $max\_objs = l_1 * l_2 * k$ objects that can be stored in the cache. Extensions to this formula could produce relatively balanced locality-preserving directory structures.

2.4.2 SQUIDM

One approach to reduce disk I/O is to circumvent the file system abstractions and store objects into a large memory-mapped file [22]. Disk space of the memory-mapped file is allocated once and access to the file are entirely managed by the virtual memory system. This has the following advantages:

Naming Stored objects are identified by the offset into the memory-mapped file which directly translates into a virtual memory address. This by-passes the overhead of translating file names into inodes and maintaining and storing those inodes.

Allocation The memory-mapped file is allocated once. If the file is created on a new file system, the allocated disk space is minimally fragmented which allows high utilization of disk bandwidth. As long as the file does not change in size, the allocated disk space will remain unfragmented. This one-time allocation also by-passes file system block and fragment allocation overhead [1]. Notice that memory-mapped files does not prevent *internal fragmentation*, *i.e.* the possible fragmentation of the content of the memory-mapped file due to application-level data management of the data stored in memory-mapped files. Since we assume infinite caches, internal fragmentation is not an issue here. See section 3 for the management of finite memory-mapped caches.

Paging Disk I/O is managed by virtual memory which takes advantage of hardware optimized for paging. The smallest unit of disk I/O is a system page instead of the size of the smallest stored object.

Thus, we expect that memory-mapping will benefit us primarily in the access of small objects by eliminating the opening and closing of small files. Most operating systems have limits on the size of memory-mapped files, and care must be taken to appropriately choose the objects to store in the limited space available. In the cache architecture SQUIDM we therefore chose the system page size (8K Byte) as upper limit. Over 70% of all object references are less or equal than 8K bytes (see figure 1 which is based on our HTTP traces). Objects larger than 8K Bytes are cached the same way as in SQUID.

To retrieve an object from a memory-mapped file we need to have its offset into the memory-mapped file and its size. In SQUIDM offset and size of each object are stored in in-memory meta-data. Instead of keeping track of the actual size of an object we defined five *segment sizes* (512, 1024, 2048, 4,096, or 8,192 Bytes). This reduces the size information from thirteen bits down to three bits. Each object is padded to the smallest segment size. In section 3 we will show more advantages of managing segments instead of object sizes.

These padded objects are contiguously written into the mapped virtual memory area in the order in which they are first referenced (and thus missed). Our conjecture was that this strategy would translate the temporal locality of the HTTP reference stream into spatial locality of virtual memory references.

We will show that this strategy also tends to concentrate very popular objects in the first few pages of the memory-mapped file; truly popular objects will be referenced frequently enough to be at the beginning of any reference stream. Clustering popular objects significantly reduces the number of page faults since those pages tend to stay in main memory. Over time, the set of popular references may change, increasing the page fault rate.

[1] This assumes that the underlying file system is not a log structured file system. File systems that log updates to data need to continually allocate new blocks and obliterate old blocks, thereby introducing fragmentation over time.

Algorithm 1 Algorithm to pack objects without crossing system page boundaries. The algorithm accepts a list of objects sizes of ≤ 8192 Bytes and outputs a list of offsets for packing each object without crossing system page boundaries (the size of a system page is 8192 Bytes).

```
proc packer(list_object_sizes) ≡
  One offset pointer for each segment size:
  512, 1024, 2048, 4096, 8192
  freelist := [0, 0, 0, 0, 0];
  offset_list := [];
  for i := 0 to length(list_of_object_sizes) − 1 do
     size := list_of_object_sizes[i];
     Determine segment size that fits object
     for segment := 0 to 4 do
        if size ≤ 2^(9+segment) then exit fi od;
     Find smallest available segment that fits
     for free_seg := segment to 4 do
        if freelist[free_seg] > 0 ∨ free_seg = 4
          then offset := freelist[free_seg];
               if free_seg = 4
                  Set 8192-pointer to next system page
                  otherwise mark free segment as taken
                  then freelist[4] := offset + 8192
                   else freelist[free_seg] := 0 fi;
               Update freelist with what is left
               for rest_seg := segment to free_seg − 1 do
                    new_offset := offset + 2^(9+rest_seg);
                    freelist[rest_seg] := new_offset od;
               exit fi od;
     append(offset, offset_list)
  od;
  offset_list.
```

2.4.3 SQUIDML

The SQUIDML architecture uses a combination of SQUIDM for objects smaller than 8K Byte and SQUIDL for all other objects.

2.4.4 SQUIDMLA

The SQUIDMLA architecture combines the SQUIDML architecture with an algorithm to *align* objects in the memory mapped file such that no object crosses a page boundary. An important requirement of such an algorithm is that it preserves reference locality. We use a packing algorithm, shown in Algorithm 1 that for the given traces only slightly modifies the order in which objects are stored in the memory-mapped file. The algorithm insures that no object crosses page boundaries.

3 Management of Memory-mapped Web Caches

In the previous section we reasoned that storing small objects in a memory-mapped file can significantly reduce disk I/O. We assumed infinite cache size and therefore did not address replacement strategies. In this section we explore the effect of replacement strategies on disk I/O of finite cache architectures which use memory-mapped files.

Cache architectures which use the file system to cache objects to either individual files or one memory-mapped file are really two-level cache architectures: the first-level cache is the buffer cache in the primary memory and the second-level cache is the disk. However, standard operating systems generally do not support sufficient user-level control on buffer cache management to control primary memory replacement. This leaves us with the problem of replacing objects in secondary memory in such a way that disk I/O is minimized.

In the following sections we first review relevant aspects of system-level management of memory-mapped files. We then introduce three replacement algorithms and evaluate their performance.

3.1 Memory-mapped Files

A memory-mapped file is represented in the virtual memory system as a virtual memory object associated with a *pager*. A pager is responsible for filling and cleaning pages from and to a file. In older Unix systems the pager would operate on top of the file system. Because the virtual memory system and the file system used to be two independent systems, this led to the duplication of each page of a memory-mapped file. One copy would be stored in a buffer managed by the buffer cache and another in a page frame managed by the virtual memory. This duplication is not only wasteful but also leads to cache inconsistencies. Newer Unix implementations have a "unified buffer cache" where loaded virtual memory pages and buffer cache buffers can refer to the same physical memory location.

If access to a virtual memory address causes a page fault, the page fault handler is selecting a *target page* and passes control to the pager which is responsible for filling the page with the appropriate data. A *pager* translates the virtual memory address which caused the page fault into the memory-mapped file offset and retrieves the corresponding data from disk.

In the context of memory-mapped files, a page is *dirty* if it contains information that differs from the corresponding part of the file stored on disk. A page is *clean* if its information matches the information on the associated part of the file on disk. We call the process of writing dirty pages to disk *cleaning*. If the target page of a page fault is dirty it needs to be cleaned before it can be handed to the pager. Dirty pages are also cleaned periodically, typically every 30 seconds.

The latency of a disk transaction does not depend on the amount of data transferred but on disk arm repositioning and rotational delays. The file system as well as disk drivers and disk hardware are designed to minimize disk arm repositioning and rotational delays for a given access stream by reordering access requests depending on the current position of the disk arm and the current disk sector. However, reordering can only occur to a limited extent. Disk arm repositioning and rotational delays are still mainly dependent on the access pattern of the access stream and the disk layout.

Studies on file systems (*e.g.* [23]) have shown that the majority of file system access is a sequential access of logically adjacent data blocks. File systems therefore establish disk layout which is optimized for sequential access by placing logically adjacent blocks into physically adjacent sectors of the same cylinder whenever possible [21]. Thus, a sequential access stream minimizes disk arm repositioning and rotational delays and therefore reduces the latency of disk transactions.

If the entire memory-mapped file fits into primary memory, the only disk I/O is caused by periodic page cleaning and depends on the number of dirty pages per periodic page cleaning and the length of the period between page cleaning. The smaller the fraction of the memory-mapped file which fits into primary memory, the higher the number of page faults. Each page fault will cause extra disk I/O. If page faults occur randomly throughout the file, each page fault will require a separate disk I/O transaction. The larger the number of dirty pages the higher the likelihood that the page fault handler will choose dirty target pages which need to be cleaned before being replaced. Cleaning target pages will further increase disk I/O.

The challenge of using memory-mapped files as caches is to find replacement strategies that keep the number of page faults as low as possible, and that create an access stream as sequential as possible.

3.2 Cache Management

We are looking for cache management strategies which optimize hit rate but minimize disk I/O. We first introduce a strategy that requires knowledge of the entire trace. Even though this strategy is not practical it serves as an illustration on how to ideally avoid disk I/O. We then investigate the use of the most common replacement strategy, LRU and discuss its possible drawbacks. This motivates the design of a third replacement strategy which uses a combination of cyclic and frequency-based replacement.

3.3 Replacement strategies

Before we look at specific replacement algorithms it is useful to review an object replacement in terms of disk I/O. All replacement strategies are extensions of the SQUIDMLA cache architecture except the "future looking" strategy which is an extension of SQUIDML. The replacement strategies act on segments. Thus the size of an object is either 512, 1K, 2K, 4K, or 8K Bytes. For simplicity an object can replace another object of the same segment size only. We call objects to be replaced the *target object* and the page on which the target object resides, the *target object's page*. Notice that this is not the same as the *target page* which is the page to be replaced in a page fault. What disk I/O is caused by a object replacement depends on the following factors:

Whether the target object's page is loaded If the target object's page is already loaded in primary memory, no immediate disk I/O is necessary. Like in all other cases the replacement dirties the target object's page. All following factors assume that the target object's page is not loaded.

Object's size Objects of size 8K Bytes replace the entire content of the object's target page. If the object is of a smaller segment size the target object's page needs to be faulted into memory to correctly initialize primary memory.

Whether the target page is dirty If the target object's page needs to be loaded, it is written to a memory location of a target page (we assume the steady-state where loading a page requires to clear out a target page). If the target page is dirty it needs to be written to disk before it can be replaced.

The best case for a replacement is when the target object's page is already loaded. In the worst case a replacement case causes two disk I/O transactions: one to write a dirty page to disk, and another to fault in the target object's page for an object of a segment size smaller than 8K Bytes.

Beside the synchronous disk I/O there is also disk I/O caused by the periodic page cleaning of the operating system. If a replacement strategy creates a large number of dirty pages, the disk I/O of page cleaning is significant and can delay read and write system calls.

3.4 "Future-looking" Replacement

Our "future looking" strategy modifies the SQUIDML architecture to use a pre-computed placement table that is derived from *the entire trace*, including all future references. The intent is to build a "near optimal" allocation policy, while avoiding the computational complexity of implementing a perfect bin-packing algorithm, which would take non-polynomial time. The placement table is used to determine whether a reference is a miss or a hit, whether an object should be cached, and where it should be placed in the cache. We use the following heuristics to build the placement table:

1. All objects that occur in the workload are sorted by their popularity and all objects that are only referenced once are discarded, since these would never be re-referenced in the cache.
2. The remaining objects are sorted by descending order of their popularity. The packer algorithm of SQUIDMLA (see algorithm 1) is then used to generate offsets until objects cannot be packed without exceeding the cache size.
3. Objects which do not fit into the cache during the second step are then placed such that they replace the most popular object, and the time period between the first and last reference of the new object does not overlap with the time period between the first and last reference of the replaced object.

The goal of the third step is to place objects in pages that are likely to be memory resident but without causing extra misses. Objects that cannot be placed into the cache without generating extra misses to cached objects are dropped on the assumption that their low popularity will not justify extra misses to more popular objects.

3.5 LRU Replacement

The LRU strategy combines SQUIDMLA with LRU replacement for objects stored in the memory-mapped file. The advantage of this strategy is that it keeps popular objects in the cache. The disadvantage of LRU in the context of memory-mapped files is that it has no concept of collocating popular objects on one page and therefore tends to choose target objects on pages that are very likely not loaded. This has two effects: First it causes a lot of page faults since a large percentage of target objects are of smaller segment size than 8K. Second, the large number of page faults creates a large number of dirty pages which causes significant page cleaning overhead and also increases the likelihood of the worst case where a replacement causes two disk I/O transactions. A third disadvantage of LRU replacement is that the selection of a target page is likely to generate a mostly random access stream instead of a more sequential access stream.

3.6 Frequency-based Cyclic (FBC) Replacement

We now introduce a new strategy we call Frequency-based Cyclic (FBC) replacement. FBC maintains access frequency counts of each cached object and a target pointer that points to the first object that it considers for replacement. Which object actually gets replaced depends on the reference frequency of that object. If the reference frequency is equal or greater than C_{max}, the target pointer is advanced to the next object of the same segment size. If the reference frequency is less than C_{max}, the object becomes the target object for replacement. After replacing the object the target pointer is advanced to the next object. If the target pointer reaches the end of the cache it is reset to the beginning. Frequency counts are aged whenever the average reference count of all objects becomes greater than A_{max}. If the average value reaches this value, each frequency count c is reduced to $\lceil c/2 \rceil$. Thus, in the steady state the sum of all reference counts stay between $N \times A_{max}/2$ and $N \times A_{max}$ (where N is the number of cached objects). The ceiling function is necessary because we maintain a minimum reference count of one. This aging mechanism follows the approach mentioned in [26, 12].

Since Web caching has a low hit rate, most cached objects are never referenced again. This in turns means that most of the time, the first object to which the target pointer points becomes the target object. The result is an almost sequential creation of dirty pages and page faults which is likely to produce a sequential access stream. Skipping popular pages has two effects. Firstly, it avoids replacing popular objects, and secondly the combination of cyclic replacement and aging factors out references to objects that are only popular for a short time. Short-term popularity is likely to age away within a few replacement cycles.

The two parameters of FBC, C_{max} and A_{max} have the following intuitive meaning. C_{max} determines the threshold below which a page is replaced if cyclic replacement points to it (otherwise it is skipped). For high C_{max} the hit rate suffers because more popular objects are being replaced. For low C_{max} more objects are skipped and the access stream becomes less sequential. With the Zipf-like distribution of object popularity, most objects are only accessed once. This allows low values for C_{max} without disturbing sequential access. A_{max} determines how often objects are aged. For high A_{max} aging takes place at a low frequency which leaves short-term-popular objects with high reference counts for a longer period of time. Low A_{max} values culls out short-term popularity more quickly but also make popular objects with a low but stable reference frequency look indistinguishable from less popular objects. Because of the Zipf-like distribution of object popularity, a high A_{max} will introduce only a relatively small set of objects that are popular for a short term only.

4 Experimental Methodology

In order to test these cache architectures we built a *disk workload generator* that simulates the part of a Web cache that accesses the file system or the virtual memory. With minor differences, the simulator performs the same disk I/O activity that would be requested by the proxy. However, by using a simulator, we simplified the task of implementing the different allocation and replacement policies and greatly simplified our experiments. Using a simulator rather than a proxy allows us to use traces of actual cache requests without having to mimic the full Internet. Thus, we could run repeatable measurements on the cache component we were studying – the disk I/O system.

The workload generators are driven by actual HTTP Web proxy server traces. Each trace entry consists of a URL and the size of the referenced object. During an experiment a workload generator sequentially processes each trace entry – the generator first determines whether a cached object exists and then either "misses" the object into the cache by writing data of the specified size to the appropriate location or "hits" the object by reading the corresponding data. Our workload generators process requests sequentially and thus our experiments do not account for the fact that the CERN and SQUID architecture allow multiple files to be open at the same time and that access to files can be interleaved. Unfortunately this hides possible file system locking issues.

We ran all infinite cache experiments on a dedicated Digital Alpha Station 250 4/266 with 512M Byte main memory. We used two 4G Byte disks and one 2G Byte disk to store cached objects. We used the UFS file system that comes with Digital Unix 4.0 for all experiments except those that calibrate the experiments in this paper to those in earlier work. The UFS file system uses a block size of 8192 Bytes and a fragment size of 1024 Bytes. For the comparison of SQUID and CERN we used Digital's Advanced File System (AdvFS) to validate our experiments with the results reported in [19].

UFS cannot span multiple disks so we needed a separate file system for each disk. All UFS experiments measured SQUID derived architectures with 16 first-level directories and 256 second-level directories. These 4096 directories were distributed over the three file systems, 820 directories on the 2G Byte disk and 1638 directories on each of the 4G Byte disks. When using memory-mapped caches, we placed 2048 directories on each 4G Byte disk and used the 2G Byte disk exclusively for the memory-mapped file. This also allowed us to measure memory-mapped-based caching separately from file-system-based caching.

We ran all finite cache experiments on a dedicated Digital Alpha Station 3000 with 64M Byte main memory and a 1.6G Byte disk. We set the size of the memory-mapped file to 160M Bytes. This size ensures ample exercise of the Web cache replacement strategies we are testing. The size is also roughly six times the size of the amount of primary memory used for memory-mapping (about 24M Bytes; the workload generator used 173M Bytes of virtual memory and the resident size stabilized at 37M Bytes). This creates sufficient pressure on primary memory to see the influence of the tested replacement strategies on buffer cache performance.

For the infinite cache experiments we used traces from Digital's corporate gateway in Palo Alto, CA, which runs two Web proxy servers that share the load by using round-robin DNS. We picked two consecutive weekdays of one proxy server and removed every non-HTTP request, every HTTP request with a reply code other than

200 (“OK”), and every HTTP request which contain “?” or “cgi-bin”. The resulting trace data consists of 522,376 requests of the first weekday and 495,664 requests of the second weekday. Assuming an infinite cache, the trace leads to a hit rate of 59%. This is a high hit rate for a Web proxy trace; it is due to the omission of non-cacheable material and the fact that we ignore object staleness.

For the finite cache experiments we used the same traces. Because we are only interested in the performance of memory-mapped files, we removed from the traces all references to objects larger than 8K Bytes since these would be stored as individual files and not in the memory-mapped file. As parameters for FBC we used $C_{max} = 3$ and $A_{max} = 100$.

Each experiment consisted of two phases: the first *warmup* phase started with a newly initialized file system and newly formatted disks on which the workload generator ran the requests of the first day. The second *measurement* phase consisted of processing the requests of the following day using the main-memory and disk state that resulted from the first phase. All measurements are taken during the second phase using the trace data of the second weekday. Thus, we can directly compare the performance of each mimicked cache architecture by the absolute values of disk I/O.

We measured the disk I/O of the simulations using AdvFS with a tool called `advfsstat` using the command `advfsstat -i 1 -v 0 cache_domain`, which lists the number of reads and writes for every disk associated with the file domain. For the disk I/O of the simulations using UFS we used `iostat`. We used `iostat rz3 rz5 rz6 1`, which lists the bytes and transfers for the three disks once per second. Unfortunately, `iostat` does not segregate the number of reads and writes.

5 Results

We first compared the results of our cache simulator to our prior work to determine that the simulator exercised the disk subsystem with similar results to the actual proxy caches. We measured the disk I/O of the two workload generators that mimic CERN and SQUID to see whether the generator approach reproduces the same relative disk I/O as observed on the real counterparts [19]. As Figure 3 shows, the disk I/O is similar when using the CERN and SQUID workload generators. This agrees with our earlier measurements showing that CERN and SQUID make similar use of the disk subsystem. The measurements were taken using the AdvFS file system because the Web proxy servers measured in [19] used that file system. The AdvFS utilities allowed us to distinguish between reads and writes. The data shows that only a third of all disk I/O are reads even though the cache hit rate is 59%.

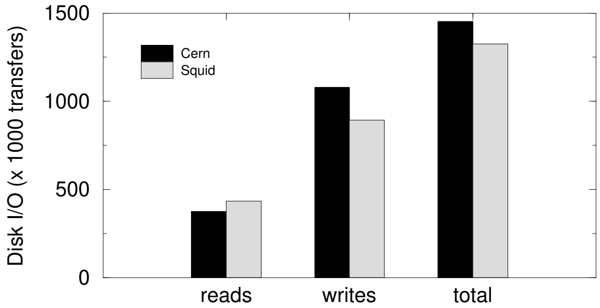


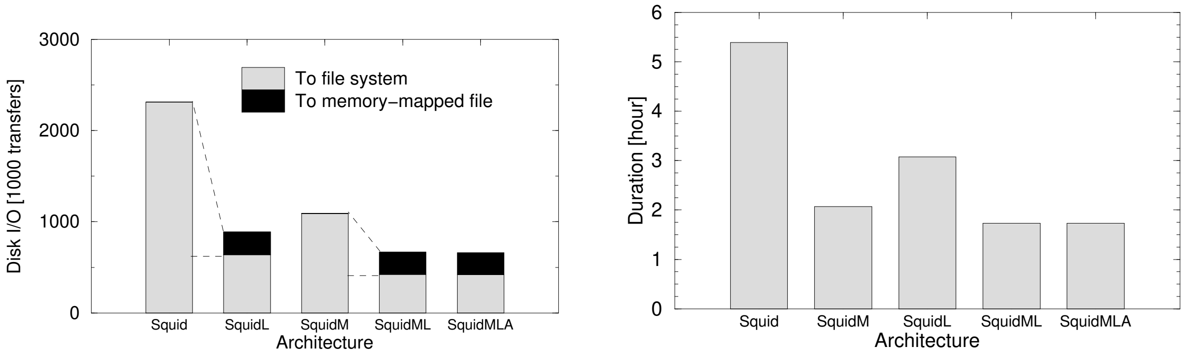
Figure 3: Disk I/O of the workload generators mimicking CERN and Squid. The measurements were taken from an AdvFS. In [19] we observed that The disk I/O of CERN and SQUID is surprisingly similar considering that SQUID maintains in-memory meta-data about its cache content and CERN does not. Our workload generators reproduce this phenomena.

Our traces referenced less than 8G Bytes of data, and thus we could conduct measurements for “infinite” caches with the experimental hardware. Figure 4 shows the number of disk I/O transactions and the duration of each trace execution for each of the architectures.

Comparing the performance of SQUID and SQUIDL shows that simply changing the function used to index the URL reduces the disk I/O by $\approx 50\%$.

By comparing SQUID and SQUIDM we can observe that memory mapping all small objects not only improves locality but produces a greater overall improvement in disk activity: SQUIDM produces 60% fewer disk I/O. Recall that SQUIDM stores all objects of size ≤ 8192 in a memory-mapped file and all larger objects in the same way as SQUID. As shown in Figure 1, about 70% of all references are to objects ≤ 8192. Thus, the remaining 30% of all references go to objects stored using the SQUID caching architecture. If we assume that these latter references account for roughly the same disk I/O in SQUIDM as in SQUID, none of the benefits come from these 30% of references. This means that there is an 85% savings generated off of the remaining 70% of SQUID’s original disk I/O. Much of the savings occurs because writes to the cache are not immediately committed to the disk, allowing larger disk transfers.

An analogous observation can be made by comparing



(a) (b)

Figure 4: Disk I/O of SQUID derived architectures. Graph (a) breaks down the disk I/O into file system traffic and memory-mapped file traffic. Graph (b) compares compares the duration of the measurement phase of each experiment

SQUIDML with SQUIDL. Here, using memory-mapping cache saves about 63% of SQUIDL's original disk I/O for objects of size $\leq$ 8192. The disk I/O savings of SQUIDM and SQUIDML are largely due to larger disk transfers that occur less frequently. The average I/O transfer size for SQUIDM and SQUIDML is 21K Bytes to the memory-mapped file, while the average transfer sizes to SQUID and SQUIDL style files are 8K Bytes and 10K Bytes, respectively.

The SQUIDMLA architecture strictly aligns segments to page boundaries such that no object spans two memory pages. This optimization would be important for purely disk-based caches, since it reduces the number of "read-modify-write" disk transactions and the number of transactions to different blocks. The results show that this alignment has no discernible impact on disk I/O. We found that SQUIDM and SQUIDML places 32% of the cached objects across page boundaries (30% of the cache hits were to objects that are crossing page boundaries).

Figure 5 confirms our conjecture that popular objects tend to be missed early. 70% of the references go to 25% of the pages to which the cache file is memory-mapped. Placing objects in the order of misses leads therefore to a higher page hit rate.

We evaluate the performance of each replacement strategy by the amount of disk I/O and the cache hit rate. As expected, the LRU replacement policy causes the highest number of disk transactions during the measurement phase. The future-looking policy shows that the actual working set at any point in time is small, and that accurate predictions of page reuse patterns would produce high hit rates on physical memory sized caches. Figure 6 show that the frequency-based cyclic replace-

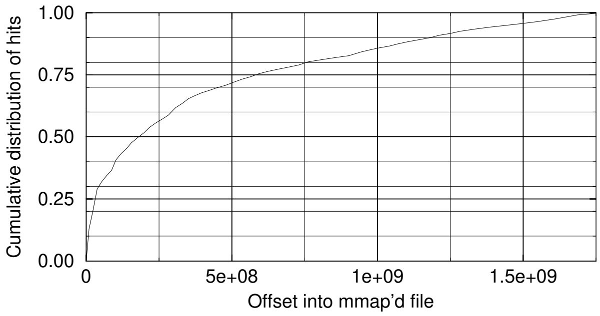


Figure 5: The cumulative hit distribution over the virtual address space of the memory-mapped cache file. 70% of the hits occur in the first quarter of the memory-mapped cache file.

ment causes less disk I/O than LRU replacement without changing the hit rate. The figure also shows the time savings caused by reduced disk I/O. The time savings are greater than the disk I/O savings which indicates a more sequential access stream where more transactions access the same cylinder and therefore do not require disk arm repositioning.

6 Related Work

There exist a large body of research work on application-level buffer control mechanisms. The external pager in Mach [32] allow users to implement paging between primary memory and disk. Cao *et al.* investigate a mechanism to allow users to manage page replacement without degrading overall performance in a multi-programmed system [7].

Glass and Cao propose and evaluate in [14] a kernel-

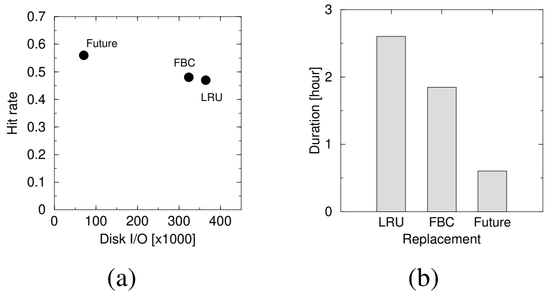


Figure 6: Disk I/O and hit rate tradeoffs of different replacement strategies. The graph (a) plots disk I/O against hit rate of the three replacement experiments. Note that *lower* x-values are better than higher ones. The graph (b) shows the duration of each experiment.

level page replacement strategy SEQ that detects long sequences of page faults and applies most-recently-used replacement to those sequences. The frequency-based cyclic web cache replacement strategy proposed above is specifically designed to generate more sequential page faults. We are currently investigating the combined performance of SEQ buffer caches and cyclic-frequency-based web caches.

7 Conclusions and Future Research

We showed that some design adjustments to the SQUID architecture can result in a significant reduction of disk I/O. Web workloads exhibit much of the same reference characteristics as file system workloads. As with any high performance application it is important to map file system access patterns so that they mimic traditional workloads to exploit existing operating caching features. Merely maintaining the first level directory reference hierarchy and locality when mapping web objects to the file system improved system the meta-data caching and reduced the number of disk I/O's by 50%.

The size and reuse patterns for web objects are also similar. The most popular pages are small. Caching small objects in memory mapped files allows most of the hits to be captured with no disk I/O at all. Using the combination of locality-preserving file paths and memory-mapped files our simulations resulted in disk I/O savings of over 70%.

Very large memory mapped caches significantly reduce the number of disk I/O requests and produce high cache hit rates. Future work will concentrate on replacement techniques that further reduce I/O from memory mapped caches while maintaining high hit rates. Our experience with the future-looking algorithm shows that there is an additional 10% reduction possible. Our experience with the LRU algorithm suggests that managing small memory mapped caches requires a tighter synchronization with the operating system memory management system. Possibilities include extensions that allow application management of pages, or knowledge of the current state of page allocation.

Our experiments do not account for overhead due to staleness of cached objects and cache replacement strategies. However, our results should be encouraging enough to motivate an implementation of some of the described cache architectures in an experimental version of SQUID.

References

[1] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Step hen Williaams, and Edward A. Fox. Caching proxies: Limitations and potentials. Available on the World-Wide Web at http://ei.cs.vt.edu/˜succeed/WWW4/WWW4.html.

[2] Jussara Almeida and Pei Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. Technical Report 1373, Computer Science Department, University of Wisconsin-Madison, April 13 1998.

[3] Virgilio Almeida, Azer Bestavros, Mark Crovella, , and Adriana de Oliveira. Characterizing Reference Locality in the WWW. In *IEEE PDIS'96: The International Conference in Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996. IEEE.

[4] Martin F. Arlitt and Carey L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *ACM Sigmetrics '96*, pages 126–137, Philadelphia, PA, May 23-26 1996. ACM Sigmetrics, ACM.

[5] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings fo the Symposium on Operating Systems Principles (SOSP)*, pages 198–212. ACM, October 1991.

[6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. Technical Report 1371, Computer Sciences Dept, Univ. of Wisconsin-Madison, April 1998.

[7] Pei Cao, Edward W. Felten, and Kai Li. Application-Controlled File Caching Policies. In *USENIX Summer 1994 Technical Conference*, 1994.

[8] Pei Cao and Sandy Irani. Cost-Aware Web Proxy Caching. In *Usenix Symposium on Internet Technologies and Systems (USITS)*, pages 193–206, Monterey, CA, USA, December 8-11 1997. Usenix.

[9] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW Client-based Traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, July 18 1995.

[10] Arjan de Vet. Squid new disk storage scheme. Available on the World Wide Web at http://www.iaehv.nl/users/devet/squid/new_store/, November 17 1997.

[11] Brad Duska, David Marwood, and Michael J. Feeley. The Measured Access Characteristics of World-Wide Web Client Proxy Caches. In *Usenix Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, USA, December 8-11 1997. Usenix.

[12] W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[13] Stewart Forster. Personal Communication. Description of disk I/O in Squid 1.2, August 14 1998.

[14] Gideon Glass and Pei Cao. Adaptive Page Replacement Based on Memory Reference Behavior. In *ACM Sigmetrics 97*, pages 115–126, Seattle, WA, June 1997.

[15] Steven Glassman. A caching relay for the world-wide web. In *First International World-Wide Web Conference*, pages 69–76. W3O, May 1994.

[16] Steven D. Gribble and Eric A. Brewer. System design issues for Internet Middleware Services: Deductions from a large client trace. In *Usenix Symposium on Internet Technologies and Systems (USITS)*, Monterey, CA, USA, December 8-11 1997. Usenix.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (2nd edition)*. Morgan Kaufmann, San Francisco, CA, 1996.

[18] Ari Luotonen, Henrik Frystyk Nielsen, and Tim Berners-Lee. CERN httpd 3.0A. Available on the World-Wide Web at http://www.w3.org/pub/WWW/Daemon/, July 15 1996.

[19] Carlos Maltzahn, Kathy Richardson, and Dirk Grunwald. Performance Issues of Enterprise Level Proxies. In *SIGMETRICS '97*, pages 13–23, Seattle, WA, June 15-18 1997. ACM.

[20] Marshall K. McKusick, William N. Joy, Samual J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[21] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, Reading, Massachusetts, 1996.

[22] Jeff Mogul. Personal communication. Idea to mmap a large file for objects less or equal than system page size, August 1996.

[23] John Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. Technical Report UCB/CSD 85/230, University of California, Berkeley, April 1985.

[24] Kathy J. Richardson. I/O Characterization and Attribute Caches for Improved I/O System Performance. Technical Report CSL-TR-94-655, Stanford University, Dec 1994.

[25] Luigi Rizzo and Lorenzo Vicisano. Replacement Policies for a Proxy Cache. Technical Report RN/98/13, University College London, Department of Computer Science, 1998.

[26] John T. Robinson and Murthy V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *SIGMETRICS'90*, pages 134–142, Boulder, CO, May 22-25 1990. ACM.

[27] Alex Rousskov and Valery Soloviev. A Performance Study of the Squid Proxy on HTTP/1.0. *to appear in World-Wide Web Journal*, Special Edition on WWW Characterization and Performance Evaluation, 1999.

[28] Alex Rousskov, Valery Soloviev, and Igor Tatarinov. Static Caching. In *2nd International Web Caching Workshop (WCW'97)*, Boulder, Colorado, June 9-10 1997.

[29] Squid Users. Squid-users mailing list archive. Available on the World-Wide Web at http://squid.nlanr.net/Mail-Archive/squid-users/hypermail.html, August 1998.

[30] Duane Wessels. Squid Internet Object Cache. Available on the World-Wide Web at http://squid.nlanr.net/Squid/, May 1996.

[31] Stephen Williams, Marc Abrams, Charles R. Standridge, Ch aleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *ACM SIGCOMM*, pages 293–305, Stanford, CA, August 1996.

[32] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *11th Symposium on Operating systems Principles*, pages 63–76, November 1987.

[33] George Kingsley Zipf. Relative Frequency as a Determinant of Phonetic Change. *reprinted from Harvard Studies in Classical Philiology*, XL, 1929.