Web-Conscious Storage Management for Web Proxies

Evangelos P. Markatos, Dionisios N. Pnevmatikatos,
Michail D. Flouris, and Manolis G.H. Katevenis,
Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385, Heraklio, Crete, GR-711-10 GREECE
http://archvlsi.ics.forth.gr markatos@csi.forth.gr

Abstract

Many proxy servers that run on today's computers are limited by their file I/O needs. Even if the proxy is configured with sufficient I/O hardware, the file system software often fails to provide the available bandwidth to the proxy processes. Although specialized file systems may offer a significant improvement and overcome these limitations, we believe that user-level disk management on top of industry standard file systems can offer comparable performance. In this paper we study the overheads associated with file I/O in web proxies, we investigate their underlying causes, and we propose *web-conscious storage management*: a set of techniques that exploit the unique reference characteristics of web-page accesses in order to allow web proxies to overcome file I/O limitations. Using realistic trace-driven simulations we show that the proxy's secondary storage I/O throughput can be improved by a factor of 18, enabling a single-disk proxy to serve around 500 (URL-get) operations per second. We demonstrate the applicability of our approach by implementing Foxy, a web proxy which incorporates our web-conscious storage management techniques. Our experimental evaluation suggests that Foxy outperforms traditional proxies such as SQUID by more than a factor of 7 under heavy load.

1 Introduction

World Wide Web proxies are being increasingly used to provide Internet access to users behind a firewall and to reduce wide-area network traffic by caching frequently used URLs. Given that web traffic still increases exponentially, web proxies are one of the major mechanisms to reduce the overall traffic at the core of the Internet, protect network servers from traffic surges, and improve the end user experience. Today's typical web proxies usually run on UNIX-like operating systems and their associated file systems. While UNIX-like file systems are widely available and highly reliable, they often result in poor performance for web proxies. For instance, Rousskov and Soloviev [35] observed that disk delays in web proxies contribute about 30% toward total hit response time. Mogul [28] observed that the disk I/O overhead of caching turns out to be much higher than the latency improvement from cache hits at the web proxy at Digital Palo Alto firewall. Thus, to save the disk I/O overhead the proxy is typically run in non-caching mode.

These observations should not be surprising, because UNIX-like file-systems are optimized for general-purpose workloads [27, 30, 21], while web proxies exhibit a distinctly different workload. For example, while read operations outnumber write operations in traditional UNIX-like file systems [3], web accesses induce a write-dominated workload [25]. Moreover, while several common files are frequently updated, most URLs are rarely (if ever at all) updated. In short, traditional UNIX-like file access patterns are different from web access patterns, and therefore, file systems optimized for UNIX-like workloads do not necessarily perform well for web-induced workloads.

In this article we study the overheads associated with disk I/O for web proxies, and propose Web-Conscious Storage Management (WebCoSM), a set of techniques designed specifically for high performance. We show that the single most important source of overhead is associated with storing each URL in a separate file, and we propose methods to aggregate several URLs per file. Once we reduce file management overhead, we show that the next largest source of overhead is associated with disk head movements due to file write requests in widely scattered disk sectors. To improve write throughput, we propose a file space allocation algorithm inspired from log-structured file systems [34]. Once write operations proceed at maximum speed, URL read operations emerge

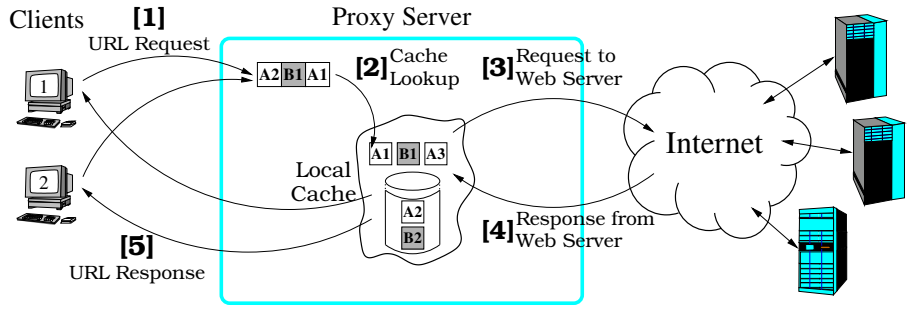


Figure 1: **Typical Web Proxy Action Sequence.**

as the next largest source of overhead. To reduce the disk read overhead we identify and exploit the locality that exists in URL access patterns, and propose algorithms that cluster several read operations together, and reorganize the layout of the URLs on the file so that URLs accessed together are stored in nearby file locations.

We demonstrate the applicability of our approach by implementing Foxy, a web proxy that incorporates our WebCoSM techniques that successfully remove the disk bottlenecks from the proxy's critical path. To evaluate the performance of our approach we use a combination of trace-driven simulation and experimental evaluation. Using simulations driven by real traces we show that the file I/O throughput can be improved by a factor of 18, enabling a single disk proxy to serve around 500 (URL-get) operations per second. Finally, we show that our implementation outperforms traditional proxies such as SQUID by more than a factor of 7 under heavy load.

We begin by investigating the workload characteristics of web proxies, and their implications to the file system performance. Then, in section 2 we motivate and describe our WebCoSM techniques, and in section 3 we present comprehensive performance results that show the superior performance potential of these techniques. We verify the validity of the WebCoSM approach in section 4 by outlining the implementation of a proof-of-concept proxy server and by evaluating its performance through experiments. We then compare the findings of this paper to related research results in section 5, discuss issues regarding WebCoSM in section 6 and summarize our findings in section 7.

2 Web-Conscious Storage Management Techniques

Traditional web proxies frequently require a significant number of operations to serve each URL request. Consider a web proxy that serves a stream of requests originating from its clients. For each request it receives, the proxy has to look-up its cache for the particular URL. If the proxy does not have a local copy of the URL (cache miss), it requests the URL from the actual server, saves a copy of the data in its local storage, and delivers the data to the requesting client. If, on the other hand, the proxy has a local copy of the URL (cache hit), it reads the contents from its local storage, and delivers them to the client. This action sequence is illustrated in Figure 1.

To simplify storage management, traditional proxies [42], [8] store each URL on a separate file, which induces several file system operations in order to serve each request. Indeed, at least three file operations are needed to serve each URL miss: (i) an old file (usually) needs to be deleted in order to make space for the new URL, (ii) a new file needs to be created in order to store the contents of this URL, and (iii) the new file needs to be filled with the URLs contents. Similarly, URL hits are costly operations, because, even though they do not change the file's data or meta-data, they invoke one file read operation, which is traditionally expensive due to disk head movements. Thus, for each URL request operation, and regardless of hit or miss, the file system is faced with an intensive stream of file operations.

File operations are usually time consuming and can easily become a bottleneck for a busy web proxy. For instance, it takes more than 20 milliseconds to create even an empty file, as can be seen in Figure 2 [1]. To make matters worse, it takes another 10 milliseconds to delete this empty file. This file creation and deletion cost approaches 60 milliseconds for large files (10 Kbytes). Given that for each URL-miss the proxy should create and delete a file, the proxy will only be able to serve one URL miss every 60 milliseconds, or equivalently, less than

[1] To quantify performance limitations of file system operations, we experimented with the HBENCH-OS file system benchmark [6]. HBENCH-OS evaluates file creation/deletion cost by creating 64 files of a given size in the same directory and then deleting them in reverse-of-creation order. We run HBENCH-OS on an UltraSparc-1 running Solaris 5.6, for file sizes of 0 to 10 Kbytes. Our results, plotted in Figure 2, suggest that the time required to create and delete a file is very high.

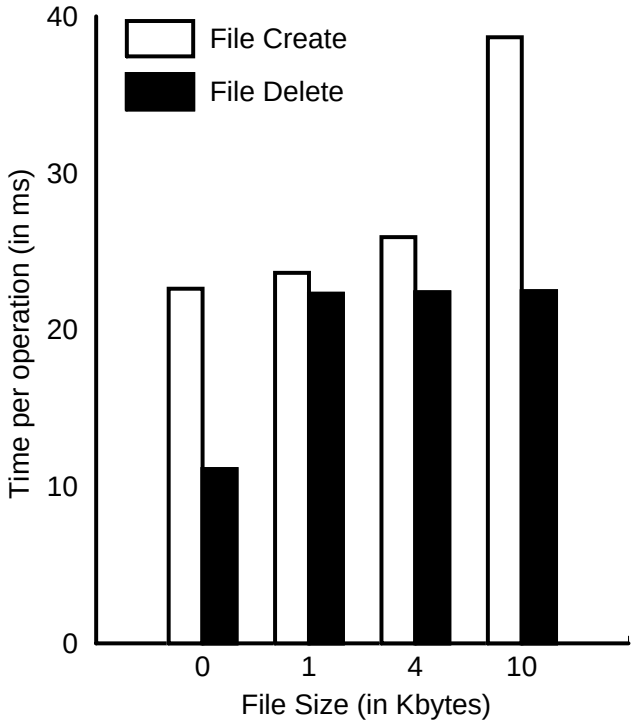


Figure 2: **File Management Overhead.** The figure plots the cost of file creation and file deletion operations as measured by the HBENCH-OS (latfs). The benchmark creates 64 files and then deletes them in the order of creation. The same process is repeated for files of varying sizes. We see that the time to create a file is more than 20 msec, while the time to delete a file is between 10 and 20 msec.

20 URL misses per second which provide clients with no more than 100-200 Kbytes of data. Such a throughput is two orders of magnitude smaller than most modern magnetic disks provide. This throughput is even smaller than most Internet connections. Consequently, the file system of a web-proxy will not be able to keep up with the proxy's Internet requests. This disparity between the file system's performance and the proxy's needs is due to the mismatch between the storage requirements needed by the web proxy and the storage guarantees provided by the file system. We address this semantic mismatch in two ways: meta-data overhead reduction, and data-locality exploitation.

2.1 Meta-Data Overhead Reduction

Most of the meta-data overhead that cripples web proxy performance can be traced to the storage of each URL in a separate file. To eliminate this performance bottleneck we propose a novel URL-grouping method (called BUDDY), in which we store all the URLs into a small number of files. To simplify space management within each of these files, we use the URL size as the grouping criterion. That is, URLs that need one block of disk space (512 bytes) are all stored in the same file, URLs that need two blocks of disk space (1024 bytes) are all stored in another file, and so forth. In this way, each file is composed of fixed-sized slots, each large enough to contain a URL. Each new URL is stored in the first available slot of the appropriate file. The detailed behavior of BUDDY is as follows:

- Initially, BUDDY creates one file to store all URLs that are smaller than one block, another file to store all URLs that are larger than a block, but smaller than two, and so on. URLs larger than a predefined threshold are stored in separate files - one URL per file.
- On a URL-write request for a given size, BUDDY finds the first free slot on the appropriate file, and stores the contents of the new URL there. If the size of the contents of the URL is above the threshold (128 Kbytes in most of our experiments), BUDDY creates a new file to store this URL only.
- When a URL needs to be replaced, BUDDY marks the corresponding slot in the appropriate file as free. This slot will be reused at a later time to store another URL.

- On a URL-read request, BUDDY finds the slot in the appropriate file and reads the content of the requested URL.

The main advantage of BUDDY is that it practically eliminates the overhead of file creation/deletion operations by storing potentially thousands of URLs per file. The URLs that occupy a whole file of their own, are large enough and represent a tiny percentage of the total number of URLs, so that their file creation/deletion overhead is not noticeable overall.

2.2 Data-Locality Exploitation

Although BUDDY reduces the file management overhead, it makes no special effort to layout data intelligently on disk in a way that improves write or read performance. However, our experience suggests that a significant amount of locality exists in the URL reference streams; identifying and exploiting this locality can result in large performance improvements.

2.2.1 Optimizing Write throughput

Both traditional proxies and our proposed BUDDY technique write new URL data in several different files scattered all over the disk, possibly requiring a head movement operation for each URL write. The number of these head movements can be significantly reduced if we write the data to the disk in a log-structured fashion. Thus, instead of writing new data in *some* free space on the disk, we continually *append* data to it until we reach the end of the disk, in which case we continue from the beginning. This approach has been widely used in log-structured file systems [5, 17, 29, 38]. However, unlike previous implementations of log-structured file systems, we use a *user-level* log-structured file management, which achieves the effectiveness of log-structured file systems on top of commercial operating systems.

Towards this end, we develop a *file-space* management algorithm (called STREAM) that (much-like log-structured file systems) streams write operations to the disk: The web proxy stores *all* URLs in a single file organized in slots of 512 bytes long. Each URL occupies an integer number of (usually) contiguous slots. URL-read operations read the appropriate portions of the file that correspond to the given URL. URL-write operations continue appending data to the file until the end of the file, in which case, new URL-write operations continue from the beginning of the file writing on free slots. URL-delete operations mark the space currently occupied the URL as free, so they can later be reused by future URL-write operations.

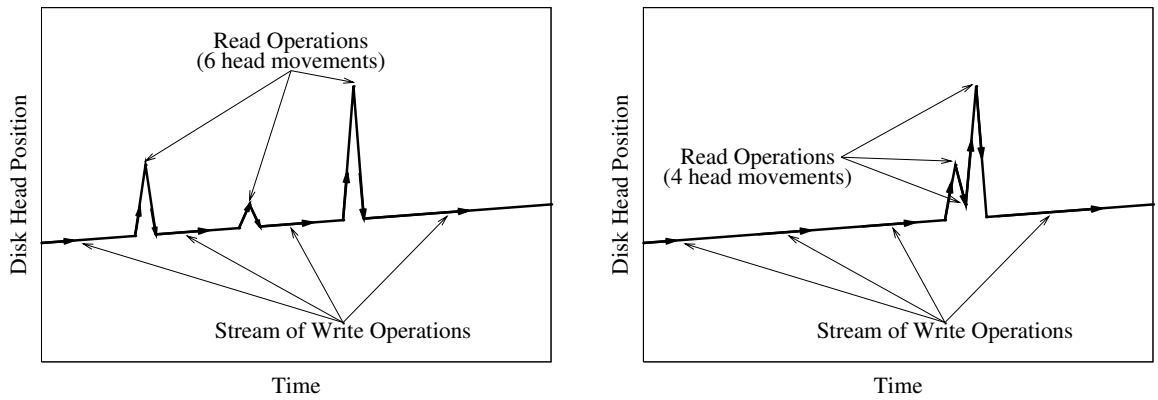
2.2.2 Improving Read Requests

While STREAM does improve the performance of write operations, *URL-read* operations still suffer from disk seek and rotational overhead, because the disk head must move from the point it was writing data to disk to the point from where it should read the data. To make matters worse, once the read operation is completed, the head must move back to the point it was before the read operation and continue writing its data onto the disk. For this reason, each read operation within a stream of writes, induces *two* head movements: the first to move the head to the reading position, and the second to restore the head in the previous writing position, resulting in a *ping-pong* effect.

To reduce this overhead, we have developed a LAZY-READS technique which extends STREAM so that it *batches* read operations. When a URL-read operation is issued, it is not serviced immediately, but instead, it is sent into an intermediate buffer. When the buffer fills up with read requests (or when a timeout period expires), the pending read requests are forwarded to the file system, sorted according to the position (in the file) of the data they want to read. Using LAZY-READS, a batch of N URL-read requests can be served with at most $N + 1$ head movements. Figure 3 illustrates the movements of the heads before and after LAZY-READS. Although LAZY-READS appear to increase the latency of URL-read operations, a sub-second timeout period guarantees unnoticeable latency increase for read operations.

2.2.3 Preserving the Locality of the URL stream

The URL requests that arrive at a web proxy exhibit a significant amount of spatial locality. For example, consider the case of an HTML page that contains several embedded images. Each time a client requests the HTML page, it will probably request all the embedded images as well. That is, the page and its embedded images are usually accessed together as if they were a single object. An indication of this relationship is that all these requests are sent to the proxy server sequentially within a short time interval. Unfortunately, these requests do not arrive



(a) Without LAZY-READS (b) With LAZY-READS

Figure 3: **Disk Head Distance.** Using the STREAM technique, the disk receives a stream of write requests to contiguous blocks interrupted only by read requests which cause the ping-pong effect. In Part (a) three read requests are responsible for six long head movements. In Part (b) the LAZY-READS technique services all three read requests together, with only four disk head movements.

sequentially to the proxy server; instead, they arrive interleaved with requests from several other clients. Therefore, web objects requested contiguously by a single client, may be serviced and stored in the proxy's disk sub-system interleaved with web objects requested from totally unrelated clients. This interleaved disk layout may result in significant performance degradation because future accesses to each one of the related web objects may require a separate disk head movement. To make matters worse, this interleaving may result in serious disk fragmentation: when the related objects are evicted from the disk, they will leave behind a set of small, non-contiguous portions of free disk space.

To recover the lost locality, we augmented the STREAM and LAZY-READS techniques with an extra level of buffers (called *locality buffers*) between the proxy server and the file system. Each locality buffer is associated with a web server, and accumulates objects that originate from that web server. Instead of immediately writing each web object in the next available disk block, the proxy server places the object into the locality buffer associated with its origin server. If no such locality buffer can be found, the proxy empties one of the locality buffers by flushing it to the disk, and creates a new association between the newly freed locality buffer and the object's web server. When a buffer fills-up, or is evicted, its contents are sent to the disk and are probably written in contiguous disk locations. Figure 4 outlines the operation of a proxy server augmented with locality buffers. The figure shows three clients, each requesting a different stream of web objects (A1-A3, B1-B2, and C1-C2). The requests arrive interleaved at the proxy server, which will forward them over the Internet to the appropriate web servers. Without locality buffers, the responses will be serviced and stored to the disk in an interleaved fashion. The introduction of the locality buffers groups the requested web objects according to their origin web server and stores the groups to the disk as contiguously as possible, reducing fragmentation and interleaving. Future read operations will benefit from the reduced interleaving through the use of prefetching techniques that will read multiple related URLs with a single disk I/O. Even future write operations will benefit from the reduced fragmentation, since they will be able to write more data on the disk in a contiguous fashion.

3 Simulation-based Evaluation

We evaluate the disk I/O performance of web proxies using a combination of simulation and experimental evaluation. In the simulation study, we model a web proxy with a storage system organized as a two level cache: a main-memory and a disk cache. Using traces obtained from busy web proxies, we drive the two-level cache simulator, which in turn generates the necessary secondary storage requests. These requests are translated into file system operations and are sent to a Solaris UFS file system and from it, to an actual magnetic disk. Therefore, in our methodology we use simulation to identify cache read, write, and delete operations, which are then sent to a real storage system in order to accurately measure their execution time. Thus, our methodology combines the ease

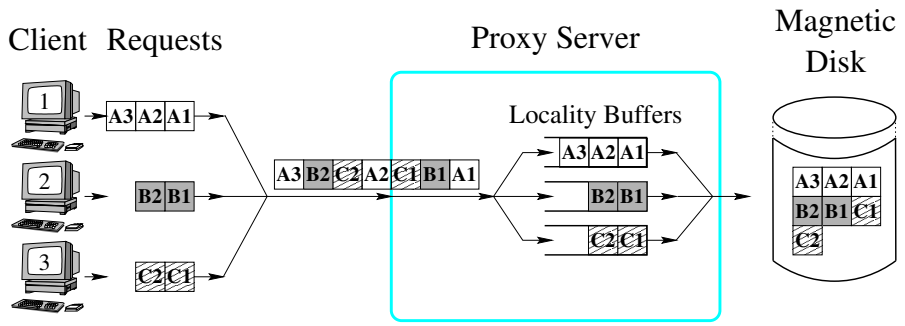


Figure 4: **Streaming into Locality Buffers.** The sequences of client requests arrive interleaved at the proxy server. The proxy server groups the requested objects in the stream into the available locality buffers, and writes the rearranged stream to the disk. For the sake of simplicity we have omitted the forwarding of the client requests over the Internet to the appropriate web server, and the web server's responses.

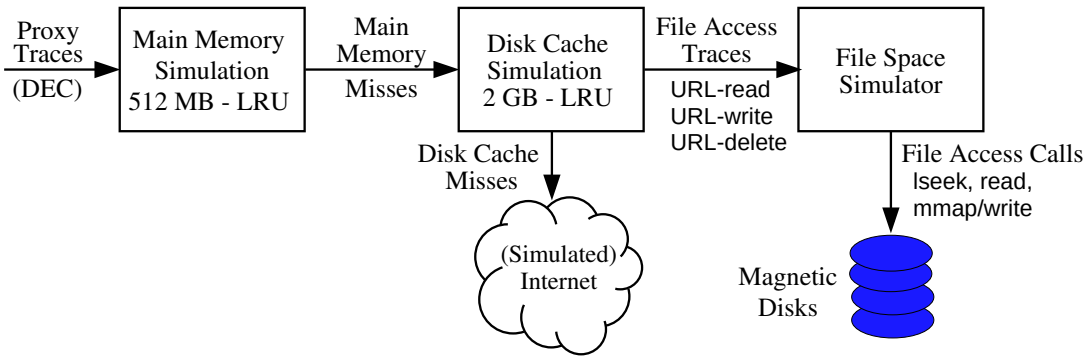


Figure 5: **Evaluation Methodology.** Traces from the DEC's web proxy are fed into a 512-Mbyte main memory LRU cache simulator. URLs that miss the main memory cache are fed into a 2-Gbyte disk LRU cache simulator. URLs that miss this second-level cache are assumed to be fetched from the Internet. These misses generate `URL-write` requests because once they fetch the URL from the Internet they save it on the disk. Second-level URL hits generate `URL-read` requests, since they read the contents of the URL from the disk. To make space for the newly arrived URLs, the LRU replacement policy deletes non-recently accessed URLs resulting in `URL-delete` requests. All `URL-write`, `URL-read`, and `URL-delete` requests are fed into a file space simulator which maps URLs into files (or blocks within files) and sends the appropriate calls to the file system.

Request Type	Number of Requests	Percentage
URL-read	42,085	4%
URL-write	678,040	64%
Main Memory Hits	336,081	32%
Total	1,058,206	100%

Table 1: **Trace Statistics**.

of trace-driven simulation with the accuracy of experimental evaluation.

3.1 Methodology

Our traces come from a busy web proxy located at Digital Equipment Corporation (`ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`). We feed these traces to a 512 Mbyte-large *first-level* main memory cache simulator that uses the LRU replacement policy [2]. URL requests that hit in the main memory cache do not need to access the second-level (disk) cache. The remaining URL requests are fed into a *second-level* cache simulator, whose purpose is to generate a trace of URL-level disk requests: *URL-read*, *URL-write*, and *URL-delete*. URL-read requests are generated as a result of a second-level cache hit. Misses in the second-level cache are assumed to contact the appropriate web server over the Internet, and save the server's response in the disk generating a URL-write request. Finally, URL-delete requests are generated when the secondary storage runs out of space, and an LRU replacement algorithm is invoked to delete unused URLs.

The generated trace of URL-read, URL-delete, and URL-write requests is sent to a file-space management simulator which forwards them to a Solaris UFS file system that reads, deletes, and writes the contents of URLs as requested. The file-space management simulator implements several secondary storage management policies, ranging from a simple "one URL per file" (SQUID-like), to STREAM, LAZY-READS, and LOCALITY-BUFFERS. The Solaris UFS system runs on top of an ULTRA-1 workstation running Solaris 5.6, equipped with a Seagate ST15150WC 4-Gbyte disk with 9 millisecond average latency, 7200 rotations per minute, on which we measured a maximum write throughput of 4.7 Mbytes per second. Figure 5 summarizes our methodology.

In all our experiments, we feed the simulator pipeline with a trace of one million URL read, write, and delete requests that are generated by 1,058,206 URL-get requests. Table 1 summarizes the trace statistics. The performance metric we use, is the total completion time needed to serve all one million requests. This time is inversely proportional to the system's throughput (operations per second) and thus is a direct measure of it. If, for example, the completion time reported is 2000 seconds, then the throughput of the system is $1058206/2000 = 529$ URL-get requests per second.

Another commonly used performance metric of web proxies, especially important for the end user, is the service latency of each URL request. However, latency (by itself) can be a misleading performance metric for our work because significant relative differences in latency can be unnoticeable to the end user, if they amount to a small fraction of a second. For example, a proxy that achieves 30 millisecond average request latency may appear twice as good as a proxy that achieves 60 millisecond average request latency, but the end user will not perceive any difference. We advocate that, as long as the latency remains within unperceivable time ranges, the proxy's throughput is a more accurate measure of the system's performance.

3.2 Evaluation

We start our experiments by investigating the performance cost of previous approaches that store one URL per file and comparing them with our proposed BUDDY that stores several URLs per file, grouped according to their size. We consider three such approaches, SINGLE-DIRECTORY, SQUID, and MULTIPLE-DIRS:

- SINGLE-DIRECTORY, as the name implies, uses a single directory to store all the URL files.
- SQUID (used by the SQUID proxy server) uses a two-level directory structure. The first level contains 16 directories (named 0..F), each of which contains 256 sub-directories (named 00..FF). Files are stored in the second level directories in a round robin manner.

[2] Although more sophisticated policies than LRU have been proposed they do not influence our results noticeably.

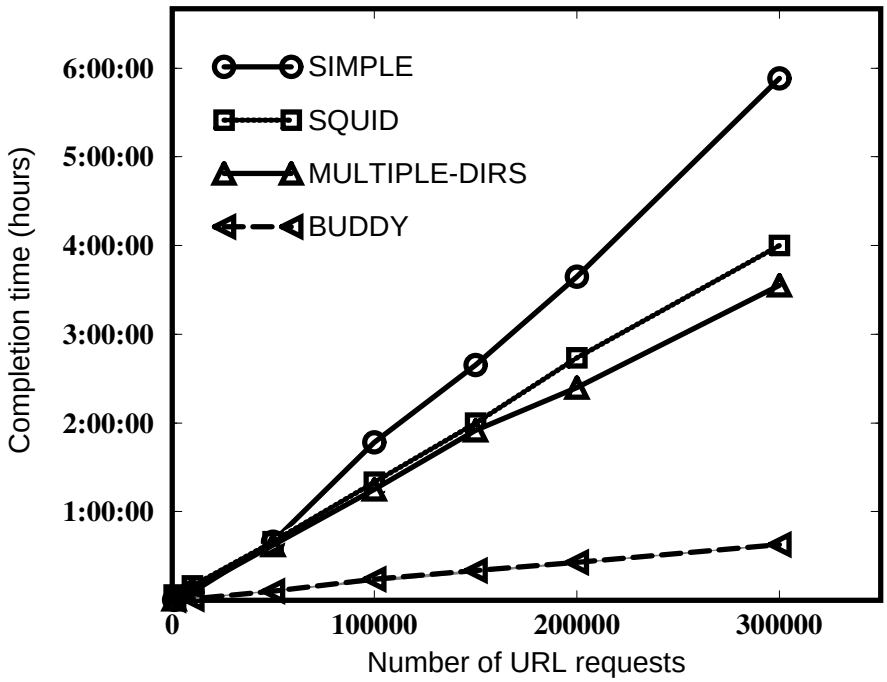


Figure 6: **File Management Overhead for Web Proxies.** The figure plots the overhead of performing 300,000 URL-read/URL-write/URL-delete operations that were generated by 398,034 URL-get requests for a 1-Gbyte large disk. It is clear that BUDDY, improves the performance considerably compared to all other approaches.

- MULTIPLE-DIRS creates one directory per server: all URLs that correspond to the same server are stored in the same directory.

Our experimental results confirm that BUDDY improves performance by an order of magnitude compared to previous approaches. Indeed, as Figure 6 shows, BUDDY takes forty minutes to serve 300,000 URL requests, while the other approaches require from six to ten times more time to serve the same stream of URL requests. BUDDY is able to achieve such an impressive performance improvement because it does not create and delete files for URLs smaller than a predefined threshold. Choosing an appropriate threshold value can be important to the performance of BUDDY. A small threshold will result in frequent file create and delete operations, while a large threshold will require a large number of BUDDY files that may increase the complexity of their management.

Figure 7 plots the completion time as a function of the threshold under the BUDDY management policy. We see that as the threshold increases, the completion time of BUDDY improves quickly, because an increasing number of URLs are stored in the same file, eliminating a significant number of file create and delete operations. When the threshold reaches 256 blocks (i.e. 128 Kbytes), we get (almost) the best performance. Further increases do not improve performance noticeably. URLs larger than 128 Kbytes should be given a file of their own. Such URLs are rare and large, so that the file creation/deletion overhead is not noticeable.

3.3 Optimizing Write Throughput

Although BUDDY improves performance by an order of magnitude compared to traditional SQUID-like approaches, it still suffers from significant overhead because it writes data into several different files, requiring (potentially long) disk seek operations. Indeed, a snapshot of the disk head movements (shown in Figure 8 taken with TazTool [9]) reveals that the disk head traverses large distances to serve the proxy's write requests. We can easily see that the head moves frequently within a region that spans both the beginning of the disk (upper portion of the figure) and the end of the disk (lower portion of the figure). Despite the clustering that seems to appear at the lower quarter of the disk and could possibly indicate some locality of accesses, the lower portion of the graph that plots the average and maximum disk head distance, indicates frequent and long head movements.

To eliminate the long head movements incurred by write operations in distant locations, STREAM stores all URLs in a single file and writes data to the file as contiguously as possible, much like log-structured file systems

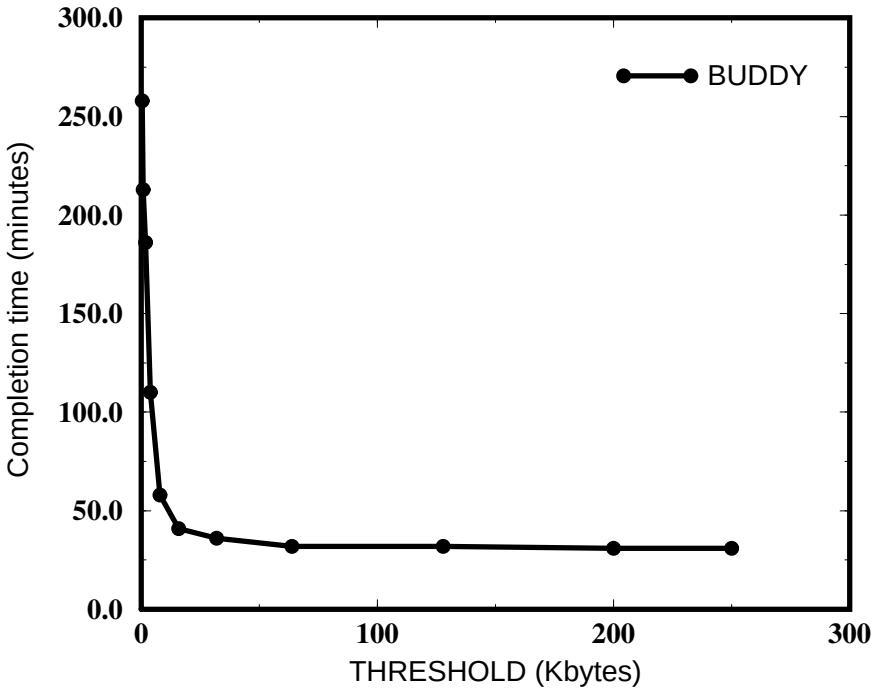


Figure 7: **Performance of** BUDDY **as a function of threshold.** The figure plots the completion time as a function of BUDDY's threshold parameter. The results suggest that URLs smaller than 64-128 Kbytes should be "buddied" together. URLs larger than that limit can be given a file of their own (one URL per file) without any (noticeable) performance penalty.

do. Indeed, a snapshot of the disk head movements (Figure 9) shows that STREAM accesses data on the disk mostly sequentially. The few scattered accesses (dots) that appear on the snapshot, are not frequent enough to undermine the sequential nature of the accesses.

Although STREAM obviously achieves minimal disk head movements, this usually comes at the cost of extra disk space. Actually, to facilitate long sequential write operations, both log-structured file systems and STREAM never operate with a (nearly) full disk. It is not surprising for log-structured file systems to operate at a disk utilization factor of 60% or even less [34], because low disk utilization increases the clustering of free space, and allows more efficient sequential write operations [3] Fortunately, our experiments (shown in Figure 10) suggest that STREAM can operate very efficiently even at 70% disk utilization, outperforming BUDDY by more than a factor of two. As expected, when the disk utilization is high (90%-95%), the performance of BUDDY and STREAM are comparable. However, when the disk utilization decreases, the performance of STREAM improves rapidly.

When we first evaluated the performance of STREAM, we noticed that even when there was always free disk space available and even in the absence of read operations, STREAM did not write to disk at maximum throughput. We traced the problem and found that we were experiencing a *small-write* performance problem: writing a small amount of data to the file system, usually resulted in *both* a disk-read *and* a disk-write operation. The reason for this peculiar behavior is the following: if a process writes a small amount of data in a file, the operating system will *read* the corresponding *page* from the disk (if it is not already in the main memory), perform the write in the main memory page, and then, at a later time, write the entire updated page to the disk.

To reduce these unnecessary read operations incurred by small writes, we developed a packetized version of STREAM, STREAM-PACKETIZER, that works just like STREAM with the following difference:

> URL-write operations are not forwarded directly to the file system - instead they are accumulated into a page-boundary-aligned one-page-long packetizer buffer, as long as they are stored contiguously to the previous URL-write request. Once the packetizer fills up, or if the current request is not contiguous to the previous one, the packetizer is sent to the file system to be written to the disk.

[3] Fortunately, recent measurements suggest that most file systems are about half-full on the average [11], and thus, log-structured approaches for file management may be more attractive than ever, especially at the embarrassingly decreasing cost of disk space [10].

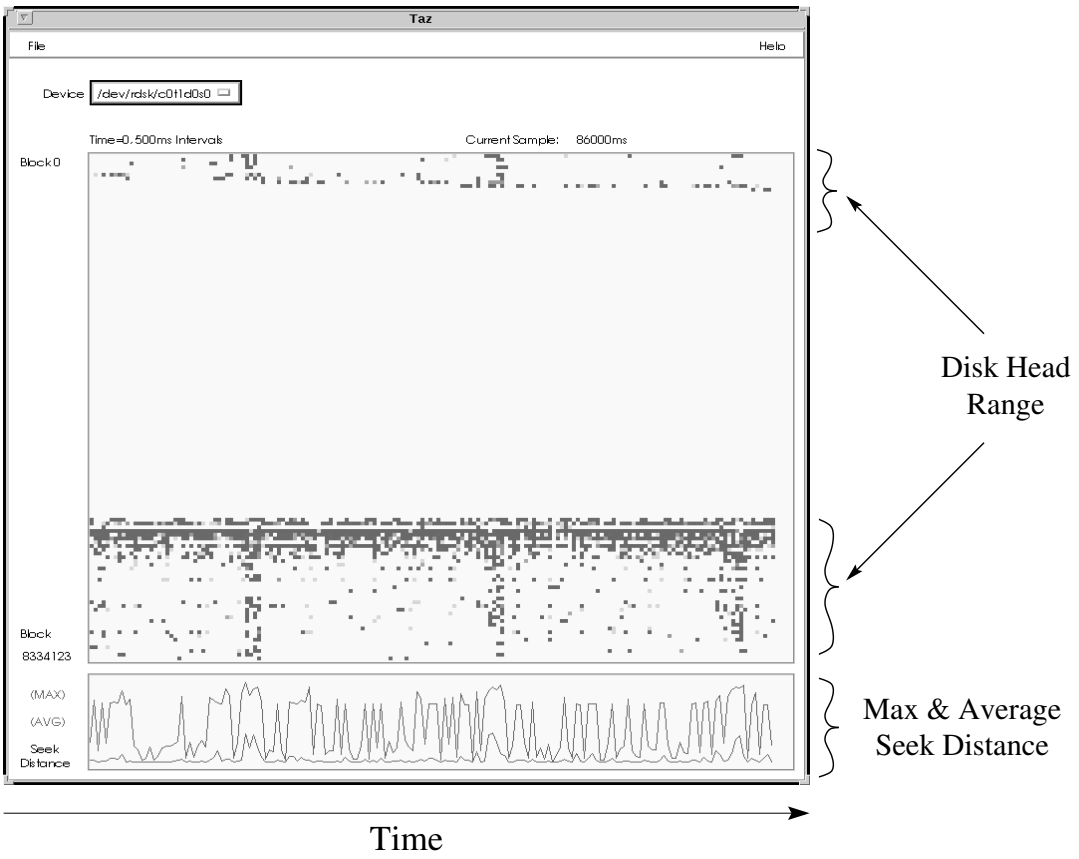


Figure 8: **Disk Access Pattern of** BUDDY**.** This snapshot was taken with TazTool, a disk head position plotting utility. The upper part of the Figure shows the span of the disk on the vertical y-axis, and time in granularity of 0.5 seconds on the horizontal x-axis, and makes a mark for each block that is accessed. The lower part plots the average and maximum disk head seek distance for each 0.5 second interval.

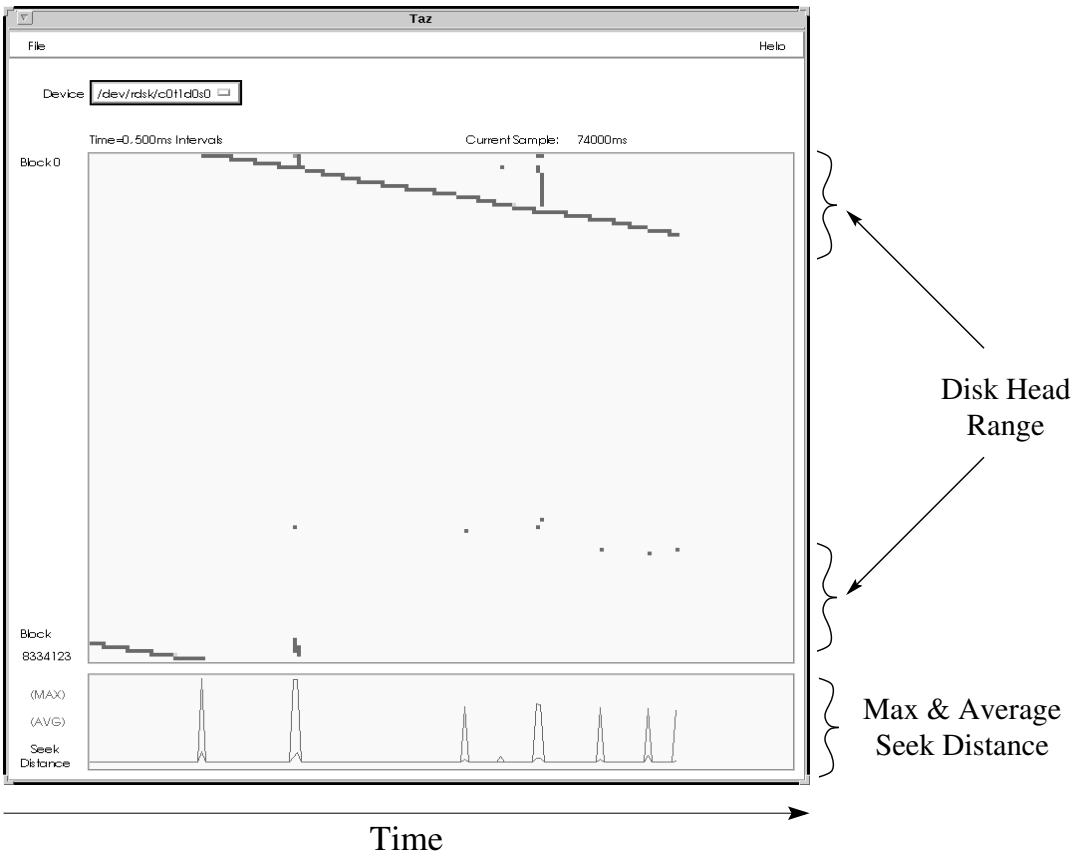


Figure 9: **Disk Access Pattern of** STREAM**.** This snapshot was taken with TazTool, a disk head position plotting utility. The upper part of the Figure shows the span of the disk on the vertical y-axis, and time in granularity of 0.5 seconds on the horizontal x-axis, and makes a mark for each block that is accessed. The lower part plots the average and maximum disk head seek distance for each 0.5 second interval.

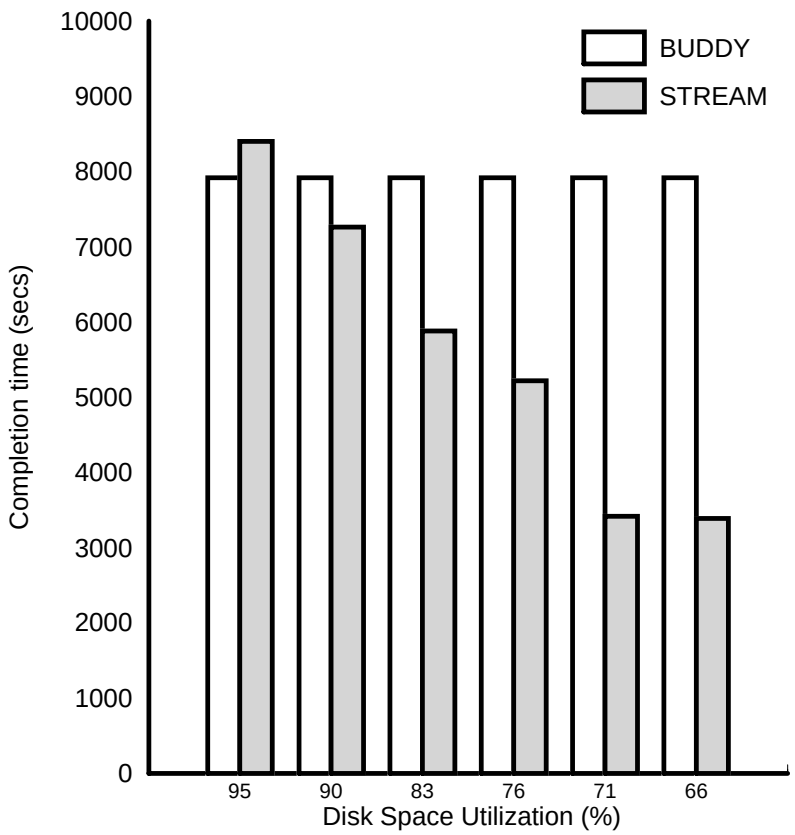


Figure 10: **Performance of** BUDDY **and** STREAM **as a function of disk (space) utilization.** The figure plots the completion time for serving 1,000,000 URL operations as a function of disk utilization. As expected, the performance of BUDDY is unaffected by the disk utilization, and the performance of STREAM improves as disk utilization decreases. When the disk utilization is around 70% STREAM outperforms BUDDY by more than a factor of two.

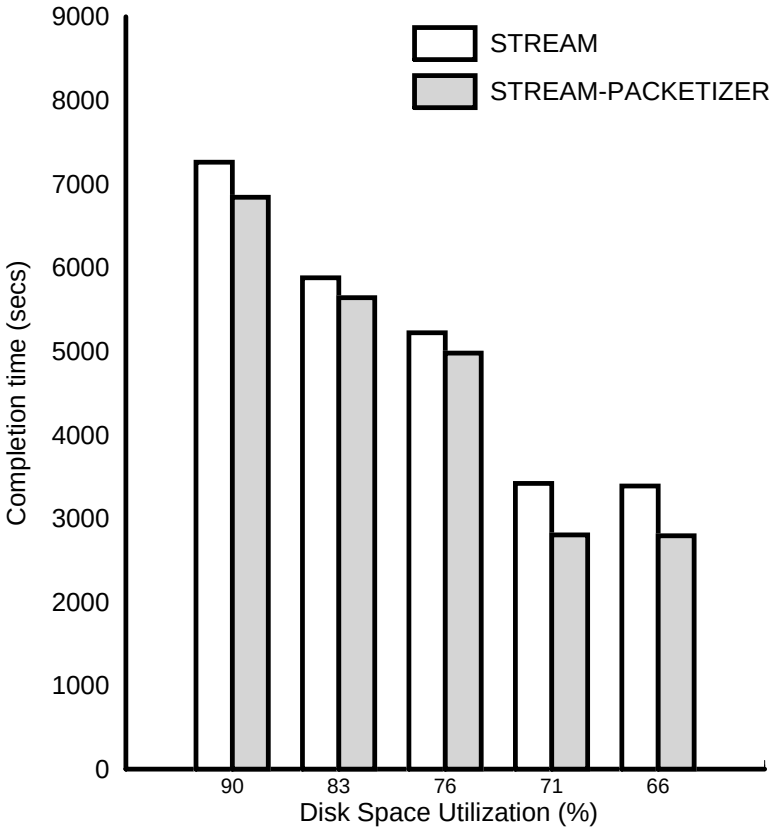


Figure 11: **Performance of** STREAM **and** STREAM-PACKETIZER **as a function of disk (space) utilization.** The figure plots the completion time for serving 1,000,000 URL operations as a function of disk utilization. STREAM consistently outperforms STREAM-PACKETIZER, by as much as 20% for low disk utilizations.

In this way, STREAM-PACKETIZER instead of sending a large number of small sequential write operations to the file system (like STREAM does), it sends fewer and larger (page size long) write operations to the file system. Figure 11 plots the performance of STREAM and STREAM-PACKETIZER as a function of disk utilization. STREAM-PACKETIZER performs consistently better than STREAM, by as much as 20% when disk utilization is low, serving one million requests in less than three thousand seconds, achieving a service rate of close to 350 URL-get operations per second.

3.4 Improving Read Requests

While STREAM improves the performance of URL-write operations, URL-read operations still suffer from seek and rotational latency overhead. A first step towards improving the performance of read operations, LAZY-READS, reduce this overhead by clustering several read operations and by sending them to the disk together. This grouping of read requests not only reduces the disk head ping-pong effect but also presents the system with better opportunities for disk head scheduling. Figure 12 shows that LAZY-READS consistently improve the performance over STREAM-PACKETIZER by about 10% [4]. Although a 10% performance improvement may not seem impressive at first glance, we believe that the importance of LAZY-READS will increase in the near future. In our experimental environment, read requests represent a small percentage (a little over 6%) of the total disk operations. Therefore,

[4] The careful reader will notice however, that LAZY-READS may increase operation latency. However, we advocate that such an increase will be unnoticeable by end-users. Our trace measurements show that STREAM-PACKETIZER augmented with LAZY-READS is able to serve 10-20 read requests per second (in addition) to the write requests. Thus LAZY-READS will delay the average read operation only by a fraction of the second. Given that the average web server latency may be several seconds long [2], LAZY-READS impose an unnoticeable overhead. To make sure that no user ever waits an unbounded amount of time to read a URL from the disk even in an unloaded system, LAZY-READS can also be augmented with a time out period. If the time out elapses then all the outstanding read operations are sent to disk.

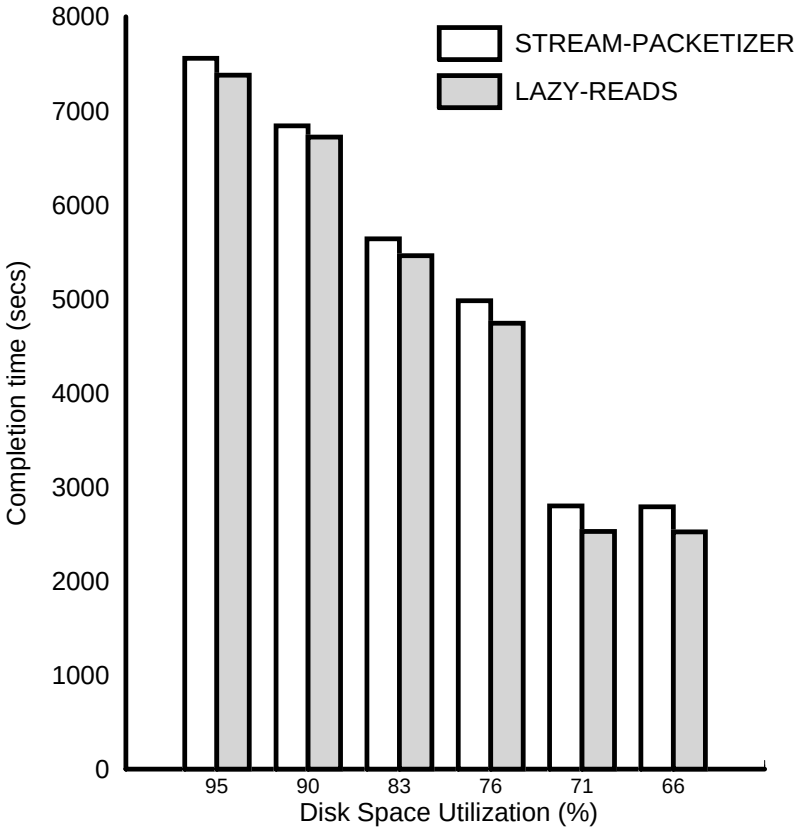


Figure 12: **Performance of** LAZY-READS. The figure plots the completion time for serving 1,000,000 URL operations as a function of 2-Gbyte disk utilization. LAZY-READS gathers reads requests ten-at-a-time and issues them all at the same time to the disk reducing the disk head movements between the write stream and the data read. The figure shows that LAZY-READS improves the performance of STREAM-PACKETIZER by 10%.

even significant improvements in the performance of read requests will not necessarily yield significant overall performance gains. With the increasing size of web caches, the expected hit rates will probably increase, and the percentage of disk read operations will become comparable to (if not higher than) the percentage of disk write operations. In this case, optimizing disk read performance through LAZY-READS or other similar techniques will be increasingly important.

3.5 Preserving the Locality of the URL stream

3.5.1 The Effects of Locality Buffers on Disk Access Patterns

To improve the performance of disk I/O even further, our locality buffers policies (LAZY-READS-LOC and STREAM-LOC) improve the disk layout by grouping requests according to their origin web server before storing them to the disk. Without locality buffers, the available disk space tends to be fragmented and spread all over the disk. With locality buffers, the available disk space tends to be clustered in one large empty portion of the disk. Indeed, the two-dimensional disk block map in Figure 13(b), shows the available free space as a long white stripe. On the contrary, in the absence of locality buffers, free space tends to be littered with used blocks shown as black dots (Figure 13(a)). Even when we magnify a mostly allocated portion of the disk (Figure 13 (a) and (b) right), small white flakes begin to appear within the mostly black areas, corresponding to small amounts of free disk space within large portions of allocated space. We see that locality buffers are able to cluster the white (free) space more effectively into sizable and square white patches, while in the absence of locality buffers, the free space if clustered into small and narrow white bands.

Figure 14 confirms that locality buffers result in better clustering of free space, plotting the average size of

Algorithm	Performance URL-get operations per second
SIMPLE	18
SQUID	27
MULTIPLE-DIRS	31
BUDDY	133
STREAM	295
STREAM-PACKETIZER	358
LAZY-READS	396
STREAM-LOC	460
LAZY-READS-LOC	495

Table 2: **Performance of traditional and Web-Conscious Storage Management techniques (in URL-get operations per second).**

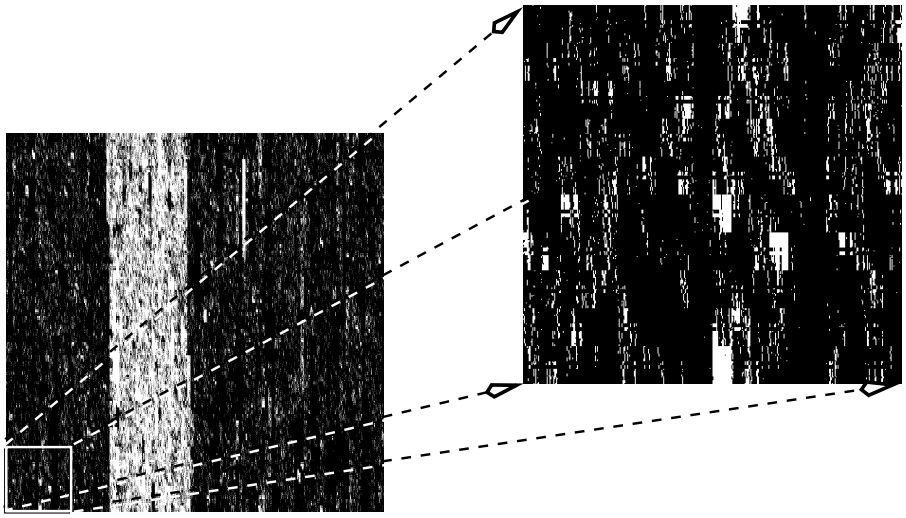
chunks of contiguous free space as a function of time. After the warm-up period (about 300 thousand requests), locality buffers manage to sustain an average free chunk size of about 145 Kbytes. On the contrary, the absence of locality buffers (STREAM) exhibits a fluctuating behavior with an average free chunk size of only about 65 Kbytes.

Locality buffers not only cluster the free space more effectively, they also populate the allocated space with clusters of related documents by gathering URLs originating from each web server into the same locality buffer, and in (probably) contiguous disk blocks. Thus, future read requests to these related web documents will probably access nearby disk locations. To quantify the effectiveness of this related object clustering, we measure the distance (in file blocks) between successive disk read requests. Our measurements suggest that when using locality buffers, a larger fraction of read requests access nearby disk locations. Actually, as many as 1,885 read requests refer to the immediately next disk block to their previous read request, compared to only 611 read requests in the absence of locality buffers (as can be seen from Figure 15). Furthermore, locality buffers improve the clustering of disk read requests significantly: as many as 8,400 (17% of total) read requests fall within ten blocks of their previous read request, compared to only 3,200 (6% of total) read requests that fall within the same range for STREAM-PACKETIZER. We expect that this improved clustering of read requests that we observed, will eventually lead to performance improvements, possibly through the use of prefetching.

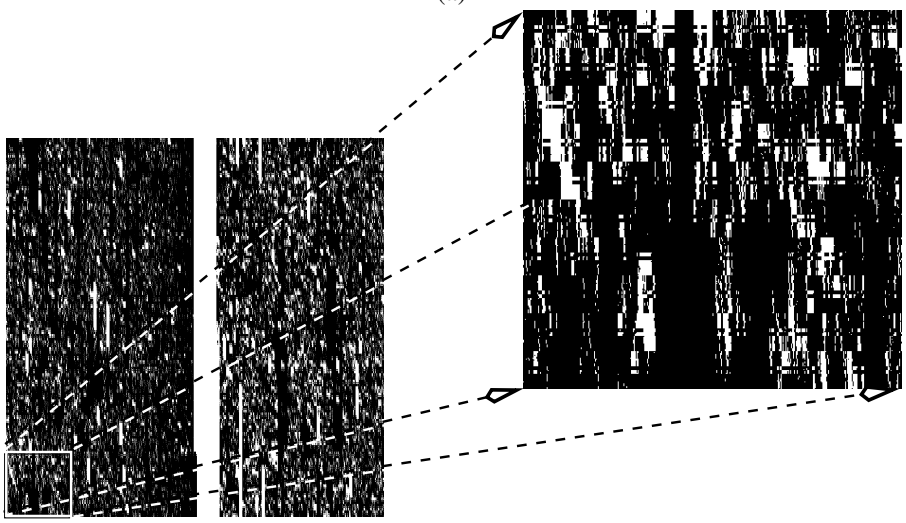
3.5.2 Performance Evaluation of LOCALITY BUFFERS

Given the improved disk layout of LOCALITY BUFFERS, we expect that the performance of LAZY-READS-LOC to be superior to that of LAZY-READS and of STREAM-PACKETIZER. In our experiments we vary the number of locality buffers from 8 up to 128; each locality buffer is 64-Kbytes large. Figure 16 shows that as few as eight locality buffers (LAZY-READS-LOC-8) are sufficient to improve performance over LAZY-READS between 5% and 20%, depending on the disk utilization. However, as the number of locality buffers increases, the performance advantage of LAZY-READS-LOC increases even further. Actually, at 76% disk utilization, LAZY-READS-LOC with 128 locality buffers performs 2.5 times better than both LAZY-READS and STREAM-PACKETIZER.

We summarize our performance results in Table 2 presenting the best achieved performance (measured in URL-get operations per second) for each of the studied techniques. We see that the Web-Conscious Storage Management techniques improve performance by more than an order of magnitude, serving close to 500 URL-get operations per second, on a single-disk system. Actually, in our experimental environment, there is little room for any further improvement. LAZY-READS-LOC-128 transfers 7.6 Gbytes of data (both to and from secondary storage) in 2,020 seconds, which corresponds to a sustained throughput of 3.7 Mbytes per second. Given that the disk used in our experiments can sustain a maximum write throughput of 4.7 Mbytes per second, we see that our WebCoSM techniques achieve up to 78% of the maximum (and practically unreachable) upper limit in performance. Therefore, any additional, more sophisticated techniques are not expected to result in significant performance improvements (at least in our experimental environment).



(a)



(b)

Figure 13: **Disk Fragmentation Map.** The Figure plots a two-dimensional disk block allocation map at the end of our simulations for STREAM (a) and LOCALITY BUFFERS (b). We plot allocated blocks with black dots and free blocks with white dots. The beginning of the disk is plotted at the lower left corner, the rest of the disk is plotted following a column-major order, and finally, the end of the disk is plotted at the top right corner.

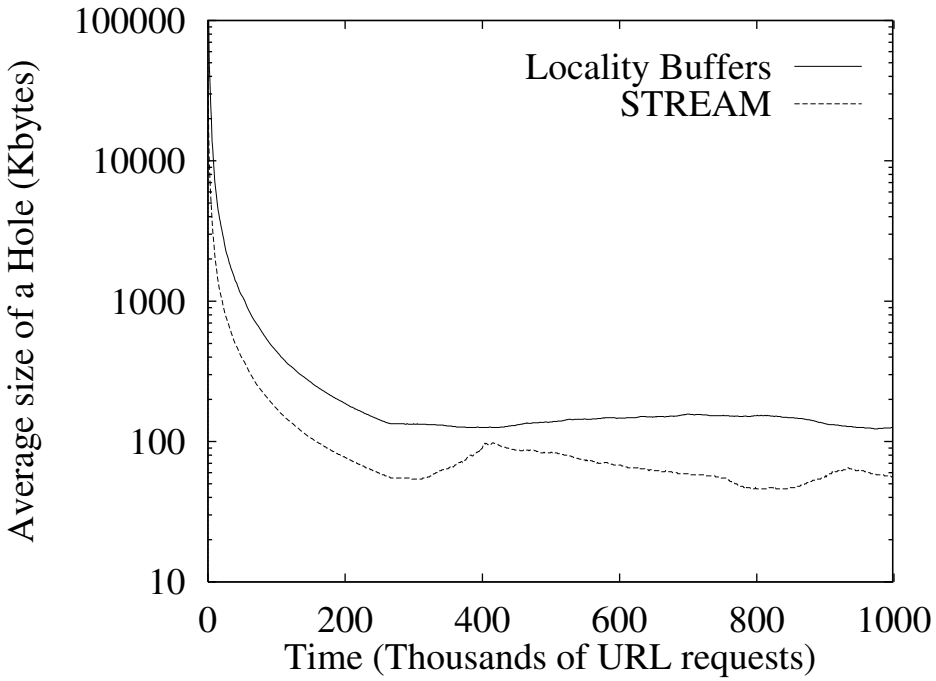


Figure 14: **Average size of free disk blocks.** The Figure plots the average size of chunks of contiguous free space as a function of time.

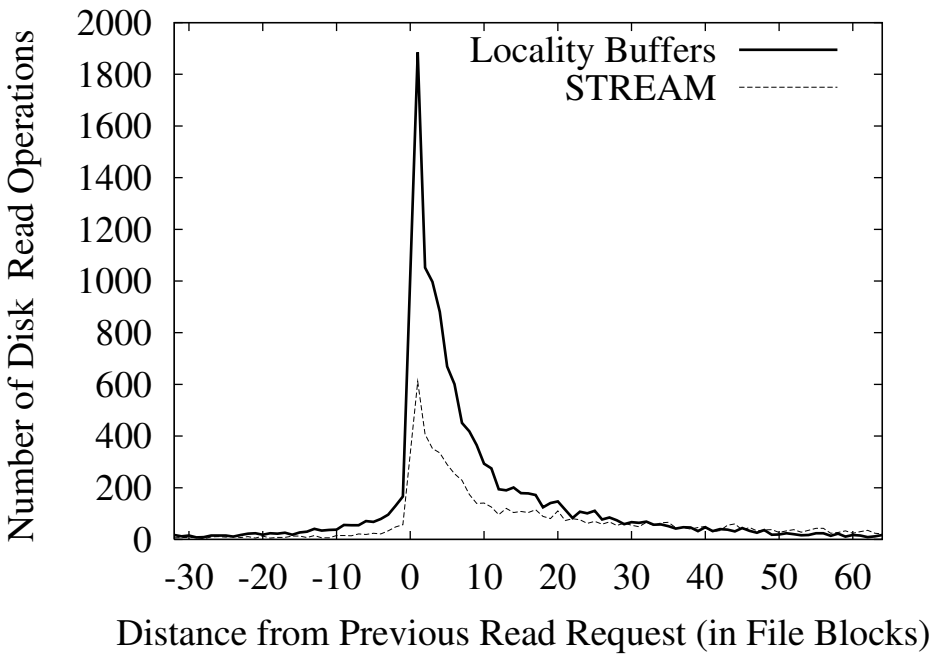


Figure 15: **Distribution of distances for Read Requests.** The Figure plots the histogram of the block distances between successive read operations for STREAM-PACKETIZER and STREAM-LOC (using 128 Locality Buffers).

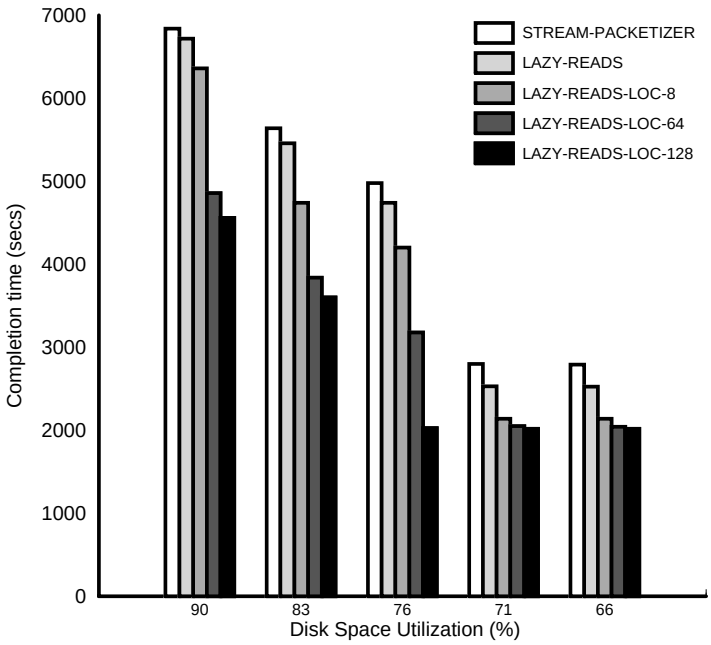


Figure 16: **Performance of** LAZY-READS-LOC. The figure plots the completion time for serving 1,000,000 URL operations as a function of disk utilization. LAZY-READS-LOC attempts to put URLs from the same server in nearby disk locations by clustering them in locality buffers before sending them to the disk.

4 Implementation

To validate our Web-Conscious Storage Management approach, we implemented a lightweight user-level web proxy server called *Foxy*. Our goal in developing Foxy was to show that WebCoSM management techniques (i) can be easily implemented, (ii) can provide significant performance improvements, and (iii) require neither extensive tuning nor user involvement.

Foxy consists of no more that 6,000 lines of C code, and implements the basic functionality of an HTTP web proxy. For simplicity and rapid prototyping, Foxy implements the HTTP protocol but not the not-so-frequently used protocols like FTP, ICP, etc. Foxy, by being first and foremost a caching proxy, uses a two level caching approach. The proxy's main memory holds the most frequently accessed documents, while the rest reside in the local disk. To provide an effective and transparent main memory cache, Foxy capitalizes on the existing file buffer cache of the operating system.

The secondary-storage system management of Foxy stores all URLs in a single file, according to the STREAM-PACKETIZER policy: URLs are contiguously appended to the disk. When the disk utilization reaches a high watermark, a cache replacement daemon is invoked to reduce the utilization below a low watermark. The replacement policy used is LRU-TH [1], which replaces the least recently used documents from the cache. In order to prevent large documents from filling-up the cache, LRU-TH does not cache documents larger than a prespecified threshold. Foxy was developed on top of Solaris 5.7, and has also being tested on top of Linux 2.2.

4.1 Design of the Foxy Web Proxy

The design and architecture of the Foxy Web Cache follows the sequence of operations the proxy must perform in order to serve a user request (mentioned here also as "proxy request" or "proxy connection"). Each user request is decomposed into a sequence of synchronous states that form a finite state machine (FSM) shown in Figure 17. Based on this FSM diagram, Foxy functions as a pipelined scheduler in an infinite loop. At every step of the loop, the scheduler selects a ready proxy request, performs the necessary operations, and advances the request into the next state.

The rectangles in Figure 17 represent the various states that comprise the processing of a request, while the clouds represent important asynchronous actions taking place between two states. Finally, lines represent transitions between states and are annotated to indicate the operations that must be completed in order to transition

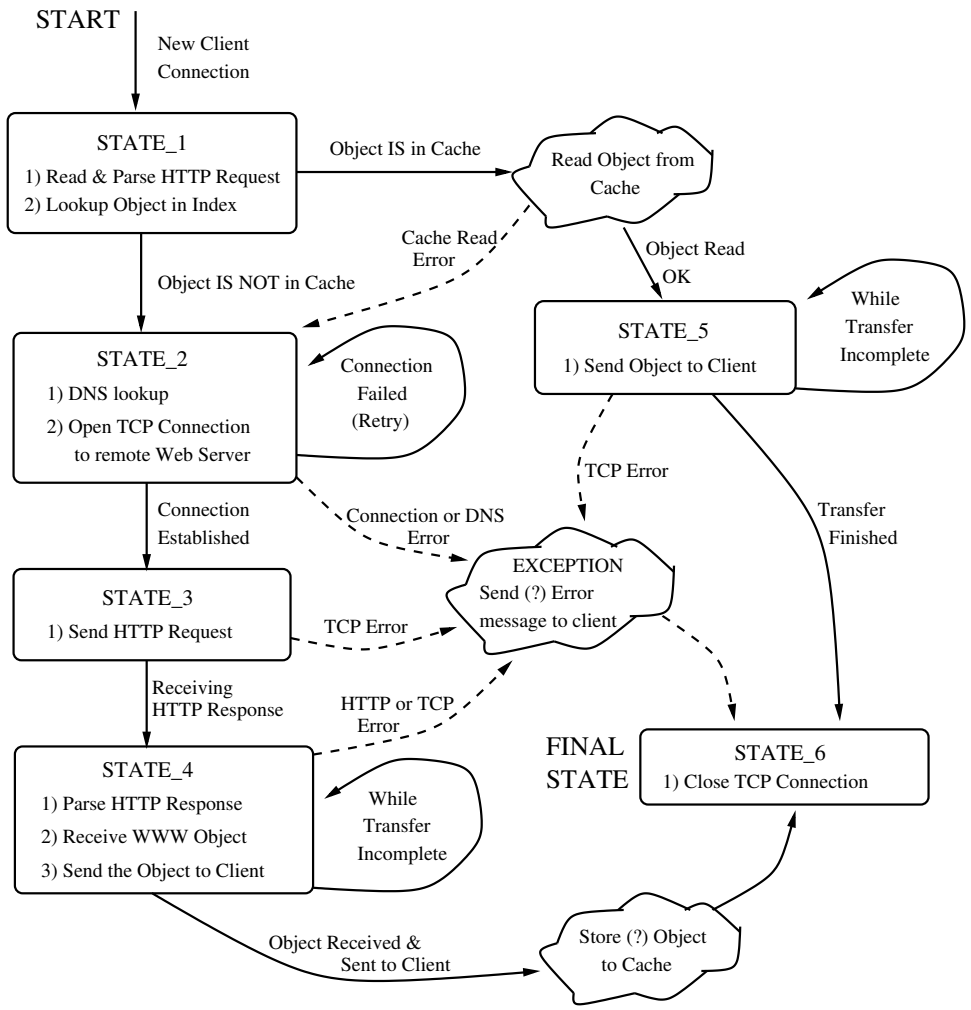


Figure 17: **The Foxy Processing Finite State Machine Diagram.**

Number of Clients	100
Number of Servers	4
Server Response Time	1 second
Document Hit Rate	40%
Request Distribution	zipf(0.6)
Document Size Distribution	exp(5 Kbytes)
Cacheable Objects	100%

Table 3: **The WebPolygraph Parameters Used in our Experiments.**

to occur. Every request begins at "STATE_1", where the HTTP request is read and parsed. Then, Foxy searches its index for the requested object. The index contains the metadata of the objects stored in the Foxy's Cache (e.g. name and size of each object, its storage location, the object's retrieval time, etc.). If the search for the requested object is successful, Foxy issues a read request to the cache. When the object is read from the cache, a transition to "STATE_5" is made, where the object is returned to the client. After a successful transfer, the client TCP/IP connection is closed in "STATE_6", and the processing of the HTTP request is completed. If the search for the requested object is unsuccessful, Foxy, in "STATE_2", performs a DNS lookup for the IP address of the remote Web server and then opens a TCP/IP connection to it. When the TCP/IP connection is established, Foxy, in "STATE_3", sends an HTTP request to the origin web server. The server response is parsed in "STATE_4", and Foxy receives the object's content over the (possibly slow) Internet connection. Foxy passes the object through to the requesting client as it receives it from the remote server, until all the object's contents are transferred. Then, the Foxy uses a cache admission policy (LRU-TH) to decide whether this object should be cached, and is so, the object's content is stored in the (disk) cache using the STREAM-PACKETIZER algorithm, and its corresponding metadata are stored in the index. Finally, the TCP connection is closed in "STATE_6", and the processing of the HTTP request is completed.

4.2 Experiments

To measure the performance of Foxy and compare it to that of SQUID, we used the Web Polygraph proxy performance benchmark (version 2.2.9), a *de facto* industry standard for benchmarking proxy servers [33]. We configured Web Polygraph with 100 clients and four web servers each responding with a simulated response time of one second. Table 3 summarizes the Web Polygraph configuration used in our experiments. We run Web Polygraph on a SUN Ultra 4500 machine with four UltraSparc-2 processors at 400MHz. The machine runs Solaris 5.7, and is equipped with a Seagate ST318203FSUN1-8G, 18-Gbyte disk. The secondary storage size that both SQUID and Foxy use is 8 Gbytes. We used the 2.2.STABLE4 version of SQUID, and we configured it according to the settings used in the Second Polygraph Bake-off [36]. To reduce the effects of a possibly slow network, we run all processes on the same computer.

The results from our experiments are presented in Figures 18, 19, and 20. Figure 18 plots the *throughput* of SQUID and Foxy, as a function of the client demands (input load) that ranges from 40 to 350 requests per second. We see that for small input load (less than 80 requests per second), the throughput of both proxies increases linearly. However, the throughput of SQUID quickly levels-off and decreases at 90 requests per second, while Foxy sustains the linear increase up to 340 requests per second, giving a factor of four improvement over SQUID.

The deficiencies of SQUID are even more pronounced in Figure 19, which plots the *average response time* achieved by the two proxies as a function of the input load. We can easily see that SQUID's latency increases exponentially for input load higher than 50 requests per second. Thus, although Figure 18 suggests that SQUID can achieve a throughput of 90 requests per second, this throughput comes at a steep increase of the request latency as seen by the end user (almost 8 seconds). The maximum throughput that SQUID can sustain without introducing a noticeable increase in latency is around 50 requests per second. On the contrary, Foxy manages to serve more than 340 requests per second without any noticeable increase in the latency. In fact, Foxy can serve up to 340 requests per second, with a user latency of about 0.7 seconds. Therefore, for the acceptable (sub-second) latency ranges, Foxy achieves almost 7 times higher throughput than SQUID.

To make matters worse, SQUID not only increases the perceived end-user latency, but it also increases the network traffic required. In fact, when the disk sub-system becomes overloaded, SQUID, in an effort to off-load the disk, may forward URL requests to the origin server, even if it has a local copy in its disk cache. This behavior

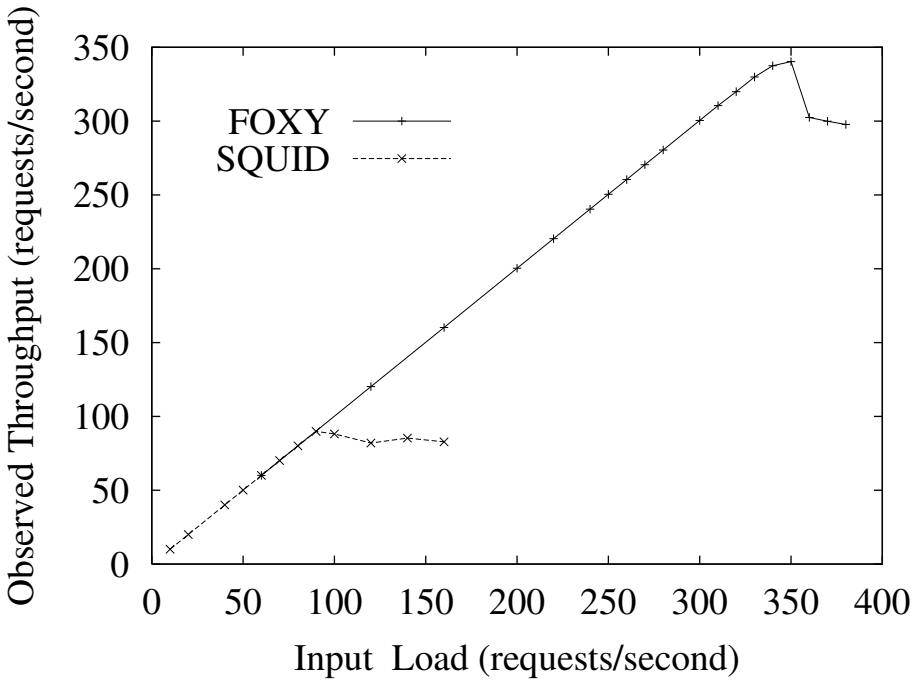


Figure 18: **Throughput of SQUID and FOXY.**

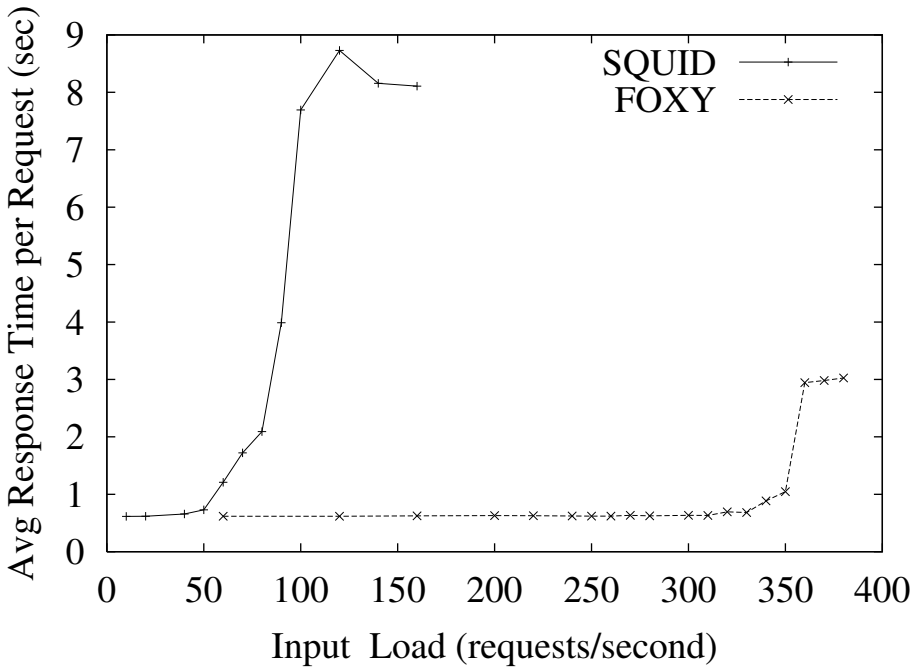


Figure 19: **Latency of SQUID and FOXY.**

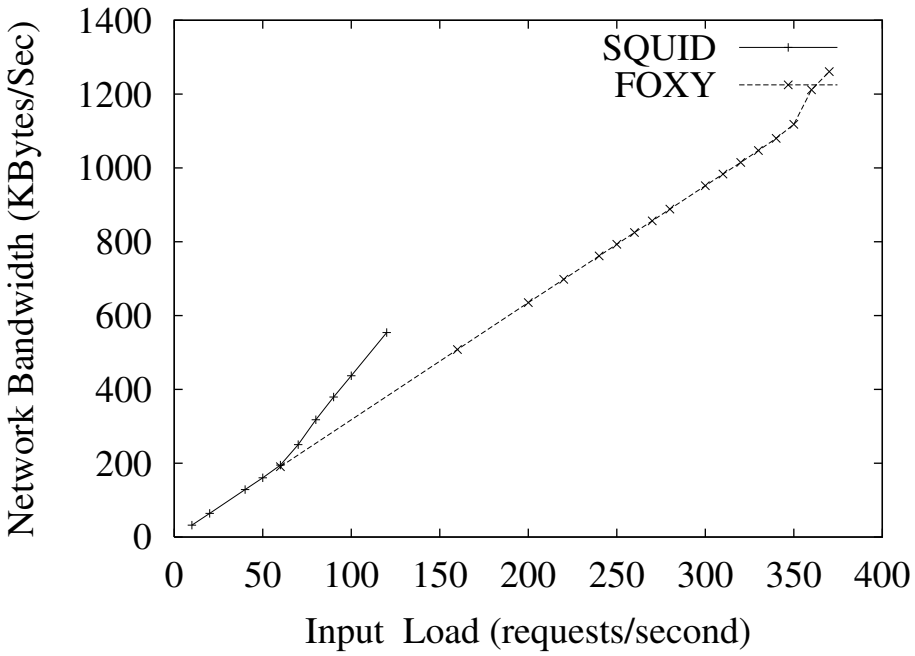


Figure 20: **Network Bandwidth Requested by SQUID and FOXY.**

may increase network traffic significantly. As Figure 20 shows when the input load is less than 50 requests per second, both SQUID and Foxy require the same network bandwidth. When the input load increases beyond 50 requests per second, the network bandwidth required by Foxy increases linearly with the input load as expected. However, the network bandwidth required by SQUID increases at a higher rate, inducing 45% more network traffic than Foxy at 110 URL requests per second.

5 Previous Work

Caching is being extensively used on the web. Most web browsers cache documents in main memory or in local disk. Although this is the most widely used form of web caching, it rarely results in high hit rates, since the browser caches are typically small and are accessed by a single user only [1]. To further improve the effectiveness of caching, proxies are being widely used at strategic places of the Internet [8, 42]. On behalf of their users, caching proxies request URLs from web servers, store them in a local disk, and serve future requests from their cache whenever possible. Since even large caches may eventually fill up, cache replacement policies have been the subject of intensive research [1, 7, 23, 26, 32, 37, 43, 44]. Most of the proposed caching mechanisms improve the end user experience, reduce the overall traffic, and protect network servers from traffic surges [15]. To improve hit rates even further and enhance the overall user experience, some proxies may even employ intelligent *prefetching* methods [4, 12, 16, 41, 31, 40].

As the Internet traffic grew larger, it was realized that Internet servers were bottlenecked by their disk subsystem. Fritchie found that USENET news servers spend a significant amount of time storing news articles in files "one file per article" [14]. To reduce this overhead he proposes to store several articles per file and to manage each file as a cyclic buffer. His implementation shows that storing several news articles per file results in significant performance improvement.

Much like news servers, web proxies also spend a significant percentage of their time performing disk I/O. Rousskov and Soloviev observed that disk delays contribute as much as 30% towards total hit response time [35]. Mogul suggests that disk I/O overhead of disk caching turns out to be much higher than the latency improvement from cache hits [28]. Thus, to save the disk I/O overhead the proxy is typically run in its non-caching mode [28].

To reduce disk I/O overhead, Soloviev and Yahin suggest that proxies should have several disks [39] in order to distribute the load among them, and that each disk should have several partitions in order to localize accesses to related data. Unfortunately, Almeida and Cao [2] suggest that adding several disks to exiting traditional web

proxies usually offers little (if any) performance improvement.

Maltzahn, Richardson and Grunwald [24] measured the performance of web proxies and found that the disk subsystem is required to perform a large number of requests for each URL accessed and thus it can easily become the bottleneck. In their subsequent work they propose two methods to reduce disk I/O for web proxies [25]:

- they store URLs of the same origin web server in the same proxy directory (SQUIDL), and
- they use a single file to store all URLs less than 8 Kbytes in size (SQUIDM).

Although our work and [25] shares common goals and approaches toward reducing disk meta-data accesses in web proxies, our work presents a clear contribution towards improving both data and meta-data access overhead:

- We propose and evaluate STREAM and STREAM-PACKETIZER, two file-space management algorithms that (much like log-structured file systems) optimize write performance by writing data contiguously on the disk.
- We propose and evaluate LAZY-READS and LAZY-READS-LOC, two methods that reduce disk seek overhead associated with read operations.

The performance results reported both in [25] and this paper, verify that meta-data reduction methods improve performance significantly. For example, Maltzahn *et al.* report that on a Digital AlphaStation 250 4/266 with 512 Mbytes of RAM and three magnetic disks, SQUID is able to serve around 50 URL-get requests per second, and their best performing SQUIDMLA approach is able to serve around 150 requests per second. Similarly, on our Sun Ultra-1 at 166 MHz with 384 Mbytes RAM and a single magnetic disk, SQUID is able to serve around 27 URL-get requests per second, while BUDDY, the simplest WebCoSM technique that reduces only meta-data overhead, achieves around 133 requests per second. Furthermore, the remaining WebCoSM techniques that improve not only meta-data, but also the data accesses, are able to achieve close to 500 URL-get requests per second.

Most web proxies have been implemented as user-level processes on top of commodity (albeit state-of-the-art) file-systems. Some other web proxies were built on top of custom-made file systems or operating systems. NetCache was build on top of WAFL, a file system that improves the performance of write operations [18]. Inktomi's traffic server uses UNIX raw devices [20]. CacheFlow has developed CacheOS, a special-purpose operating system for proxies [19]. Similarly, Novell has developed a special-purpose system for storing URLs: the Cache Object Store [22]. Unfortunately very little information has been published about the details and performance of such custom-made web proxies, and thus a direct quantitative comparison between our approach and theirs is very difficult. Although custom-made operating systems and enhanced file-systems can offer significant performance improvements, we choose to explore the approach of running a web proxy as a user-application on top of a commodity UNIX-like operating system. We believe that our approach will result in lower software development and maintenance costs, easier deployment and use, and therefore a quicker and wider acceptance of web proxies.

The main contributions of this article are:

- We study the overheads associated with file I/O in web proxies, investigate their underlying causes, and propose WebCoSM, a set of new techniques that overcome file I/O limitations.
- We identify locality patterns that exist in web accesses, show that web proxies destroy these patterns, and propose novel mechanisms that restore and exploit them.
- The applicability of our approach is shown through Foxy, a user-level web proxy implementation on top of a commercial UNIX operating system.
- Comprehensive performance measurements using both simulation and experimental evaluation show that our approach can easily achieve an order of magnitude performance improvement over traditional approaches.

6 Discussion

6.1 Reliability Issues

Although SQUID-like policies that store one URL per file do not perform well, they appear to be more robust in case of a system crash than our WebCoSM approach, because they capitalize on the expensive reliability provided by the file system. For example, after a system crash, SQUID can scan all the directories and files in its disk cache, and recreate a complete index of the cached objects. On the contrary, WebCoSM methods store meta-data associated

with each URL in main memory for potentially long periods of time, increasing the cache's vulnerability to system crashes. For example, STREAM stores all URLs in a single file, which contains no indication of the beginning block of each URL. This information is stored in main memory, and can be lost in a system crash. Fortunately, the seemingly lost reliability does not pose any real threat to the system's operation:

- WebCoSM methods can periodically (i.e. every few minutes) write their metadata information on safe storage, so that in the case of a crash they will only lose the work of the last few minutes. Alternatively, they can store along with each URL, its name, size, and disk blocks. In case of a crash, after the system reboots, the disk can be scanned, and the information about which URLs exist on the disk can be recovered.

- Even if a few cached documents are lost due to a crash, they can be easily retrieved from the web server where they permanently reside. Thus, a system crash does not lose information *permanently*; it just loses the *local copy* of some data (i.e. a few minutes worth) which can be easily retrieved from the web again.

6.2 Lessons Learned

During the course of this research we were called to understand the intricate behavior of a web proxy system. Some of the most interesting lessons that we learned include:

- *User requests may invoke counter-intuitive operating system actions.* For example, we observed that small write requests in STREAM surprisingly invoked disk *read* operations. In this case, there is no intuitive and direct correspondence between what the user requests and what the operating system actually does. This mismatch between the user requests and the operating system actions not only hurts performance, but also undermines the user's understanding of the system.

- *System bottlenecks may appear in places least expected.* It is a popular belief that proxies are bottlenecked by their network subsystem. On the contrary, we found that the secondary storage management system is also a major (if not more significant) bottleneck because it has not been designed to operate efficiently with web workloads. For example, traditional file systems used to serve no more than 10 concurrent users, requesting no more than 50 Kbytes per second each [3]. On the contrary, a busy web proxy (especially a country-wide proxy), may be required to serve hundreds of concurrent users requesting data totaling several Mbytes per second [13].

- *Optimizing the performance of read operations will be one of the major factors in the design of secondary storage management systems of web proxies.* Write operations can usually be implemented efficiently and proceed at disk bandwidth. On the contrary, read operations (even asynchronous ones), involve expensive disk seek and rotational latencies, which are difficult if not impossible to avoid. As disk bandwidth improves much faster than disk latency, read operations will become an increasing performance bottleneck.

- *Locality can manifest itself even when not expected.* We found that clients exhibit a significant amount spatial locality, requesting sequences of related URLs. Traditional proxies tend to destroy this locality, by interleaving requests arriving from different clients. Identifying and exploiting the existing locality in the URL stream are challenging tasks that should be continuously pursued.

7 Summary-Conclusions

In this paper we study the disk I/O overhead of world-wide web proxy servers. Using a combination of trace-driven simulation and experimental evaluation, we show that busy web proxies are bottlenecked by their secondary storage management subsystem. To overcome this limitation, we propose WebCoSM, a set of techniques tailored for the management of web proxy secondary storage. Based on our experience in designing, implementing, and evaluating WebCoSM, we conclude:

- The single largest source of overhead in traditional web proxies is the file creation and file deletion costs associated with storing each URL on a separate file. Relaxing this one-to-one mapping between URLs and files, improves performance by an order of magnitude.

- Web clients exhibit a locality of reference in their accesses because they usually access URLs in clusters. By interleaving requests from several clients, web proxies tend to conceal this locality. Restoring the locality in the reference stream results in better layout of URLs on the disk, reduces fragmentation, and improves performance by at least 30%.
- Managing the mapping between URLs and files in user level improves performance over traditional web proxies by a factor of 20 overall, leaving little room for improvement by specialized kernel-level implementations.

We believe that our results are significant today and they will be even more significant in the future. As disk bandwidth improves at a much higher rate than disk latency for more that two decades now [10], methods like WebCoSM that reduce disk head movements and stream data to disk will result in increasingly larger performance improvements. Furthermore, web-conscious storage management methods will not only result in better performance, but they will also help to expose areas for further research in discovering and exploiting the locality in the Web.

Acknowledgments

This work was supported in part by the Institute of Computer Science of Foundation for Research and Technology -Hellas, in part by the University of Crete through project "File Systems for Web servers" (1200). We deeply appreciate this financial support.

Panos Tsirigotis was a source of inspiration and of many useful comments. Manolis Marazakis and George Dramitinos gave us useful comments in earlier versions of this paper. Katia Obraczka provided useful comments in an earlier version of the paper. P. Cao provided one of the simulators used. We thank them all.

References

[1] M. Abrams, C.R.Standridge, G. Abdulla, S. Williams, and E.A. Fox. Caching Proxies: Limitations and Potentials. In *Proceedings of the Fourth International WWW Conference*, 1995.

[2] J. Almeida and P. Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. *Computer Networks and ISDN Systems*, 30:2179–2192, 1998.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proc. 13-th Symposium on Operating Systems Principles*, pages 198–212, October 1991.

[4] Azer Bestavros. Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems. In *Proceedings of ICDE'96: The 1996 International Conference on Data Engineering*, March 1996.

[5] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic Cleaning Algorithms for Log-Structured File Systems. In *Proceedings of the 1995 Usenix Technical Conference*, January 1995.

[6] A. Brown and M. Seltzer. Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture. In *Proc. of the 1997 ACM SIGMETRICS Conference*, pages 214–224, 1997.

[7] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. of the first USENIX Symposium on Internet Technologies and Systems*, pages 193–206, 1997.

[8] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *Proc. of the 1996 Usenix Technical Conference*, 1996.

[9] Adrian Cockcroft. Disk tracing revisited. *SunWorld*, January 1999.

[10] M. Dahlin. *Serverless Network File Systems*. PhD thesis, UC Berkeley, December 1995.

[11] J.R. Douceur and W.J. Bolosky. A Large-Scale Study of File System Contents. In *Proc. of the 1999 ACM SIGMETRICS Conference*, pages 59–70, 1999.

[12] D. Duchamp. Prefetching Hyperlinks. In *Proc. of the second USENIX Symposium on Internet Technologies and Systems*, October 1999.

[13] B. M. Duska, D. Marwood, and M. J. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proc. of the first USENIX Symposium on Internet Technologies and Systems*, pages 23–35, 1997.

[14] S.L. Fritchie. The Cyclic News Filesystem: Getting INN To Do More With Less. In *Proc. of the 1997 Systems Administration Conference*, pages 99–111, 1997.

[15] The Global Internet Project. Caching Technology: Preventing Internet Bottlenecks and Enhancing the User Experience.

[16] J. Gwertzman and M. Seltzer. The Case for Geographical Push Caching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[17] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *Proc. 14-th Symposium on Operating Systems Principles*, pages 29–43, December 1993.

[18] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proc. of the 1994 Winter Usenix Technical Conference*, pages 235–246, 1994.

[19] Cache Flow Inc. http://www.cacheflow.com.

[20] Inktomi Inc. The Sun/Inktomi Large Scale Benchmark. http://www.inktomi.com/inkbench.html.

[21] D. Kotz. Disk-Directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, 1997.

[22] R. Lee and G. Tomlinson. Workload Requirements for a Very High-Capacity Proxy Cache Design. In *Proceedings of the Web Caching Workshop*, March 1999.

[23] P. Lorenzetti, L. Rizzo, and L. Vicisano. Replacement Policies for a Proxy Cache, 1998. http://www.iet.unipi.it/˜ luigi/research.html.

[24] C. Maltzahn, K. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proc. of the 1997 ACM SIGMETRICS Conference*, pages 13–23, 1997.

[25] C. Maltzahn, K. Richardson, and D. Grunwald. Reducing the Disk I/O of Web Proxy Server Caches. In *Proc. of the 1999 Usenix Technical Conference*, 1999.

[26] E.P. Markatos. Main Memory Caching of Web Documents. *Computer Networks and ISDN Systems*, 28(7-11):893–906, 1996.

[27] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, AUG 1984.

[28] Jeffrey C. Mogul. Speedier Squid: A Case Study of an Internet Server Performance Problem. *;login: The USENIX Association Magazine*, 24(1):50–58, 1999.

[29] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[30] J. Ousterhout, H. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. 10-th Symposium on Operating Systems Principles*, pages 15–24, Orcas Island, WA, Dec 1985.

[31] V.N. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *SIGCOMM Computer Communication Review*, 26:22–36, 1996.

[32] J.E. Pitkow and M. Recker. A Simple, Yet Robust Caching Algorithm Based on Dynamic Access Patterns. In *Proceedings of the Second International WWW Conference*, 1994.

[33] Web Polygraph. Web Polygraph Proxy Performance Benchmark. http://polygraph.ircache.net/.

[34] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc. 13-th Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[35] A. Rousskov and V. Soloviev. On Performance of Caching Proxies. In *Proc. of the 1998 ACM SIGMETRICS Conference*, 1998.

[36] A. Rousskov, D. Wessels, and G. Chisholm. The Second IRCache Web Cache Bake-off. http://bakeoff.ircache.net/.

[37] P. Scheuearmann, J. Shim, and R. Vingralek. A Case for Delay-Conscious Caching of Web Documents. In *6th International World Wide Web Conference*, 1997.

[38] M. Seltzer, M. K. McKusick, K. Bostic, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the 1995 Winter Usenix Technical Conference*, San Diego, CA, January 1993.

[39] V. Soloviev and A. Yahin. File Placement in a Web Cache Server. In *Proc. 10-th ACM Symposium on Parallel Algorithms and Architectures*, 1998.

[40] Joe Touch. Defining High Speed Protocols : Five Challenges and an Example That Survives the Challenges. *IEEE JSAC*, 13(5):828–835, June 1995.

[41] Stuart Wachsberg, Thomas Kunz, and Johnny Wong. Fast World-Wide Web Browsing Over Low-Bandwidth Links, 1996. http://ccnga.uwaterloo.ca/˜ sbwachsb/paper.html.

[42] D. Wessels. Squid Internet Object Cache, 1996. http://squid.nlanr.net/Squid/.

[43] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla, and E.A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. of the ACM SIGCOMM 96*, 1996.

[44] Roland P. Wooster and Marc Abrams. Proxy Caching that Estimates Page Load Delays. In *6th International World Wide Web Conference*, 1997.