Secondary Storage Management for Web Proxies

Evangelos P. Markatos Manolis G.H. Katevenis
Dionisis Pnevmatikatos Michail Flouris*

Computer Architecture and VLSI Systems Division
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

World-Wide Web proxies are being increasingly used to provide Internet access to users behind a firewall and to reduce wide-area network traffic. Recent results suggest that disk I/O is increasingly becoming the limiting factor for the performance of web proxies. In this paper we study the overheads associated with disk I/O for web proxies, and propose secondary storage management alternatives that improve performance. We use a combination of experimental evaluation and simulation based on traces from busy web proxies. We show that web proxies experience significant overheads due to disk I/O. We propose several file management methods that reduce the disk I/O overhead overhead by a factor of 25 overall, resulting in a single-disk service rate that exceeds 500 (URL-get) operations per second.

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. Recent results suggest that disk I/O overhead is becoming an increasingly important bottleneck for the performance of web proxies. For example, Rousskov and Soloviev [33] observed that disk delays contribute about 30% toward total hit response time. Mogul states that their observations of the web proxy at Digital Palo Alto firewall suggest the disk I/O overhead of caching turns out to be much higher than the latency improvement from cache hits [27]. Thus, to save the disk I/O overhead the server is typically run in its non-caching mode.

In this paper we study the overheads associated with disk I/O for web proxies, and propose secondary storage management alternatives that improve performance. We show that the single most important source of overhead is associated with storing each URL in a separate file. File system operations (like a file creation followed by a file deletion) may easily take up to 50 milliseconds (aggregate), even on modern hardware. Given that the median size of a cached file is 3KBytes [33], and that for each URL typical web proxies create one file (to store the contents of the URL) and delete another file (to free space), then the rate at which a web proxy can store data to disk is 3KBytes every 50 msec, or roughly 60 KBytes/sec, a rate of execution that is orders of magnitude lower than the data transfer rates that current disks can sustain. This data rate is even lower than most Internet connections. To alleviate this file management overhead we propose a storage management method called BUDDY that stores several URLs per file. BUDDY identifies URLs of similar sizes ("buddies") and stores them in the same file. BUDDY reduces file management overhead by storing several URLs per file and reduces fragmentation by storing similar-sized URLs in each file.

Once we reduce file management overhead using BUDDY, we show that the next largest source of overhead is associated with disk head movements due to file write requests which write data in widely scattered places over the disk space. To improve write throughput, we propose a file space allocation algorithm (called STREAM) inspired from log-structured file systems [32]. STREAM stores all URLs in a single file contiguously (if possible). STREAM reduces disk seek and rotational overheads and manages to perform write operations at maximum speed. Once write operations proceed at maximum speed, URL read operations emerge as the next largest source of overhead. To reduce the disk read overhead we propose algorithms that cluster several read operations together (LAZY-READS) (in order to reduce the disk head seek time), and organize the layout of the URLs on the file so that URLs accessed together are

*All the authors are also with the Unversity of Crete.

stored in nearby file locations (locality buffers).

To evaluate the performance of our approach we use a mix of trace-driven simulation and experimental evaluation. Traces from the DEC's web proxy are fed into a 512-Mbyte main memory LRU cache simulator [7]. 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. Our results suggest that BUDDY achieves an order of magnitude improvement over traditional web proxy approaches, STREAM achieves a factor of 2-3 improvement over BUDDY, and locality buffers achieves a 20%-150% improvement over STREAM.

The rest of the paper is organized as follows: Section 2 surveys previous work. Section 3 presents our algorithms, and evaluates their performance. Section 4 summarizes and concludes the paper.

2 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 is the least effective, since it rarely results in large hit rates [1]. To improve cache hit rates, caching proxies are used [8, 39]. Proxies employ large caches which they use to serve stream of requests coming from a large number of users. Since even large caches may eventually fill up, cache replacement policies have been the subject of recent research [1, 7, 20, 23, 31, 34, 40, 41]. Sophisticated caching mechanisms usually improve the observed user latency and reduce network traffic. Some caches may even employ intelligent *prefetching* methods to improve the hit rate even further [3, 4, 13, 24, 38, 29, 37].

Recently, it was realized that web proxies spend a significant percentage of their time doing disk I/O. Rousskov and Soloviev observed that disk delays contribute 30% towards total hit response time [33]. Mogul suggests that disk I/O overhead of disk caching turns out to be much higher than the latency improvement from cache hits [27]. To reduce disk I/O overhead Soloviev and Yahin suggest that proxies should have several disks, and that each disk should have several partitions. Using sophisticated write distribution policies Soloviev and Yahin are able to spread requests over several disks and to cluster requests on the same partition to avoid long seek delays [36]. Scott Fritchie found that USENET News servers spend a significant amount of time storing articles in files "one file per article" [12]. 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 URLs per file results in significant performance improvement. Maltzahn, Richardson and Grunwald [21] measured the performance of web proxies. With regard to disk I/O they measured that several disk accesses are needed for each URL request (in the average). This implies 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 [22]:

- they preserve locality of the http reference stream by storing files of the same web server in the same proxy directory (SQUIDL) and
- they use a single file to store all objects less than 8K in size (SQUIDM).

It seems that both the authors of this paper as well as Maltzahn, Richardson and Grunwald have independently discovered similar key ideas that reduce the disk overhead of a web proxy. For example, SQUIDL and SQUIDM of [22] use similar locality and file management principles to our algorithms (MULTIPLE-DIRS and BUDDY respectively). These common ideas reduce the file management (meta-data) overhead associated with storing URLs in a file system. However, our work also presents a clear contribution towards improving data (not meta-data) access overhead:

- We propose and evaluate STREAM and STREAM-PACK, 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 (URL hit) operations.

As a result our algorithms improve on the performance of SQUID by a factor more than 25, while [22] reports performance improvements of a factor close to 5.

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 [16]. Inktomi's traffic server uses UNIX raw devices [18]. CacheFlow has developed CacheOS, a special-purpose operating system for proxies [17]. 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 file-systems can result in the best performance, we chose to explore the approach of running a web proxy as a user-application on top of a commodity operating system. Our approach will result in higher portability and more widespread use of web proxies.

The main contributions of our work in the area of disk I/O of web proxies are:

- We identify as the single largest source of overhead the storage of each URL in a separate file. We show the extend of this overhead, and propose a novel file management algorithm (BUDDY) to reduce it by an order of magnitude.
- We identify as the next single largest source of overhead the cost associated with file write operations. We propose a file space management approach (inspired by log-structured file systems) called STREAM that groups several independent write requests into long sequential writes that minimize disk head movement.
- Once write operations proceed at maximum speed (with the use of STREAM-based algorithms), read operations (although fewer in number) represent the next single largest source of overhead. We propose novel methods (LAZY-READS and LAZY-READS-LOC) that reduce the disk head movements associated with disk read operations.

3 Evaluation

3.1 Methodology

To evaluate the disk I/O performance of web proxies we use a combination of simulation and experimental evaluation as shown in Figure 1. We use traces from a SQUID web proxy used at Digital Equipment Corporation (`ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`). We feed these traces to a *first-level* main memory cache simulator [7]. The simulated main-memory is 512 Mbytes large and replaces URLs using the Least-Recently Used (LRU) replacement policy. [1] URL requests that miss the main memory cache are fed into a *second-level* cache simulator that simulates the magnetic disk cache. Second-level hits read the contents of the URL from the disk and generate a *URL-read* request. Second-level misses are assumed to be sent to the appropriate web server over the Internet, and the server's response is saved in the disk generating a *URL-write* request. When the disk runs out of space, an LRU replacement algorithm is invoked, which may delete old files generating a *URL-delete* request. URL-delete requests are also generated when new versions of cached files are requested. The generated URL-read, URL-delete, and URL-write requests are sent to a file-space management simulator which forwards them to a Solaris UFS file system which reads, deletes, and writes URLs as requested. In all our experiments we report the total time (completion time) to serve the first million URL-read/URL-write/URL-delete operations. [2] The completion time reported in our experiments is inversely proportional to the throughput (operations per second) of the system 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. It is possible to argue however, that, besides throughput, latency is also an important metric, especially for the end user. However, latency (by itself) can be a misleading performance metric for our work. For example, suppose that proxy server A has a 15 msec average operation response latency and manages to sustain 50 operations per second, while server B has a 30 msec average operation response latency and manages to sustain 100 operations per second. Although latency may favor server A, most implementations will probably prefer to use server B, since it achieves higher throughput and its increase in latency is not noticeable by most humans. For this reason, our performance results focus on server throughput while making sure that our policies do not increase latency noticeably. This happens in most cases without particular effort because our policies interact with disks that operate in the millisecond range, while the typical world-wide web latency is in the second range (3-4 seconds per request in the average when lightly loaded, and more than 10 seconds per request when heavily loaded [2, 19]).

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

[2] The file space management simulator is fed with 1058206 URL-get requests that generate one million URL-read/URL-write/URL-delete operations. These 1058206 URL-get requests result in 338081 (32%) main memory hits, 42085 (4%) secondary memory hits, and 678040 (64%) misses, which result in 678040 URL-write operations, 42085 URL-read operations, and 279875 URL-delete operations.

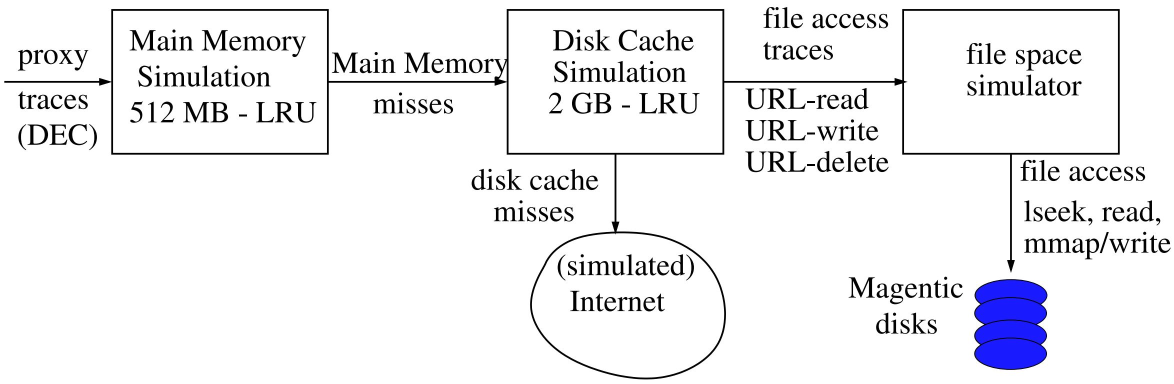

Figure 1: **Simulation Methodology.** Traces from the DEC's web proxy are fed into a 512-Mbyte main memory LRU cache simulator. URLs that miss the main memory cache are fed into a 2-Gbyte disk LRU cache simulator. URLs that miss this second-level cache are assumed to be fetched from the Internet. These 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.

Our experimental environment consists of an ULTRA-1 workstation running Solaris 5.6, equipped with a Seagate ST15150WC 4Gbyte disk with 9 ms average latency, 7200 rotations per minute, on which we measured a maximum of 4.7 Mbytes per second write throughput.

3.2 Workload Characterization

In this first set of experiments we will demonstrate that the traffic sent to the disk subsystem of a web proxy is dominated by write requests. Figure 2 plots the number of URL-read and URL-write operations that are sent to the file system of the proxy server (for 5 million URL-get requests). The number of URL-write operations is around 3 million, and decreases slowly with disk size (since larger disks imply fewer URL misses). The number of URL-read requests is less than half a million and increases with disk size (since larger disks imply more URL hits). [3]

Figure 2 suggests that the number of URL-read operations is significantly smaller than the number of URL-write operations. This is because URL-write operations correspond to second-level URL misses which can be quite large, while URL-read operations correspond to URLs that miss the first-level cache but hit in the second-level cache, which are usually a small percentage.

[3] Note that the sum of URL-read and URL-write requests is in all cases 3.4 million and not 5 million as one might expect. This because the 512-Mbyte first level cache is able to achieve a 32% URL hit rate, which leaves 3.4 million URL requests for the second level cache.

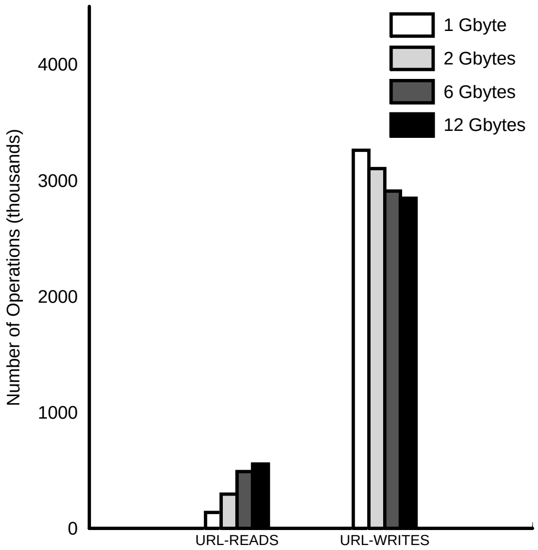

Figure 2: **File System Operations.** The figure displays the amount of URL operations (read/write) during the execution of a web proxy for various disk sizes. In all cases, write operations outnumber read operations.

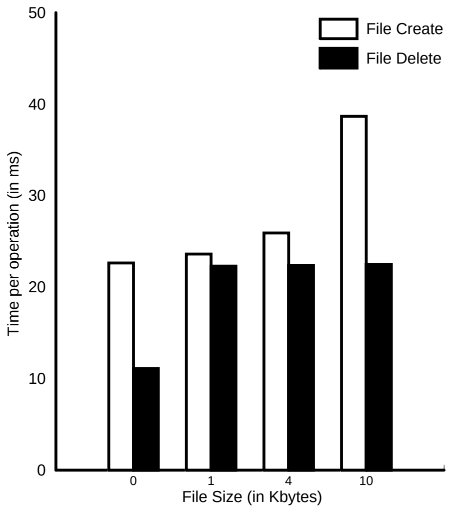

Figure 3: **File management Overhead.** The figure displays the cost of file creation and file deletion as measured by the HBENCH-OS (latfs). The benchmark creates 64 files and then deletes them in the order of creation. The same process is repeated for files of varying sizes. We see that the time to create a file is more than 20 msec. The time to delete a file is between 10 and 20 msec. The time to create and delete a 4-Kbyte file is close to 50 msec, which implies that this file system can create no more than 20 such files per second.

3.3 File Management Overhead

In this set of experiments we will demonstrate that the single most important overhead associated with the disk I/O of web proxies is the result of storing each URL in a separate file. To measure this file system overhead we use the HBENCH-OS benchmark [6] that creates 64 files in a directory and then deletes them in reverse-of-creation order. After the benchmark is repeated a number of times, the average time to do file operation, as well as its standard deviation are reported. In all but two cases the standard deviation was less than 1% away of the average time, and in the two remaining cases it was less than 10%. Figure 3 plots the results. We see that the time to create a file is more than 20 milliseconds - even when the file is empty. The time to create a 10-Kbyte-large file is close to 40 milliseconds. The time to delete a file is more than 20 milliseconds (for non-zero-sized files). If we use a benchmark that creates/deletes more than 64 files, these times will go up since the traditional UNIX directory lookup takes time linear in the directory length. Published research results using similar benchmarks agree with our measurements [26].

For each URL fetched from a web server, a typical web proxy needs to create a file to store the contents of the URL. When the disk subsystem runs out of space, for each new file created (in the average) an old file will have to be deleted (to make free space). Thus, for each URL fetched from a web server, one file is created and one file is deleted. Figure 3 suggests that the cost of a file creation and a file deletion is about 50 msec, which implies that a web proxy that incurs such a file creation/deletion cost can fetch from the network (and store in the local disk) no more than 20 URLs per second. Given that the median size of a cached file is only about 3 Kbytes long [33], then the web proxy can serve data at a rate of 3 Kbytes every 50 msec, or about 60 Kbytes per second, a throughput that is two orders of magnitude smaller than most modern magnetic disks provide. This throughput is even smaller than most Internet connections. Thus, it is obvious why researchers observe that " the disk I/O overhead of caching turns out to be much higher than the latency improvement from cache hits" [27].

3.4 File Management

Most publicly available popular web proxies (including Squid [39], Harvest [8], and CERN) store each URL on a separate file. These files are stored in a shallow directory hierarchy (like Squid) or in a deep directory hierarchy (like CERN and Harvest). We believe that file management can be the largest limiting factor in the performance of a web proxy. To alleviate this performance bottleneck we propose a novel file-grouping method called BUDDY. The main idea behind BUDDY is that each file may store several URLs. 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 are stored in another file, etc. Each file essentially is composed of same-sized slots. Each new URL is stored is the first free slot of the appropriate file. BUDDY behaves as follows:

- 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, another file to store all URLs that are larger than two blocks, but smaller than three, and so on, up to a predetermined number of blocks (THRESHOLD). URLs larger than this number are stored in separate files - one URL per file.
- On a file-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 a certain threshold (128 Kbytes in most of our experiments), BUDDY creates a new file to store this specific URL only. [4]

[4] The effect of this threshold on performance is studied in figure 5.

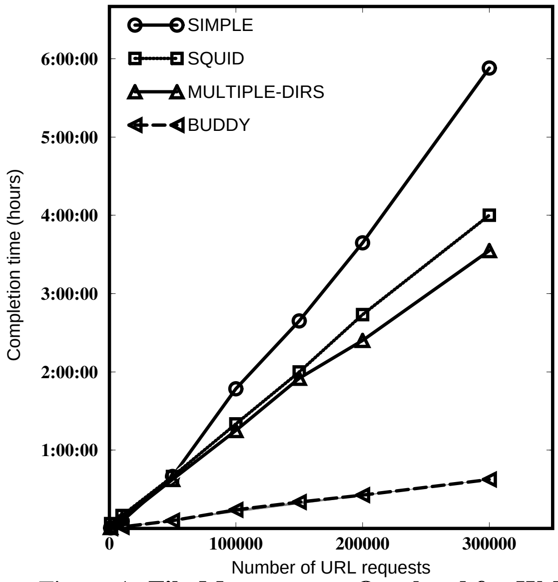

Figure 4: **File Management Overhead for Web Proxies.** The figure displays the overhead of doing 300,000 second-level cache file system operations on a 1-Gbyte disk. We see that both methods that create one file for each URL they need to store perform very bad. BUDDY, which stores several URLs per file takes roughly less than 9 msec per file operation.

- On a file-delete request, BUDDY marks the corresponding slot in the appropriate file as free. This slot will be reused at a later time by another URL of the given size.
- On a file-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. [5] The URLs that occupy a whole file of their own, represent a tiny percentage of the total number of URLs, so that their file creation/deletion overhead is not noticeable overall. One more advantage of BUDDY is that by placing same-sized URLs on the same file, it eliminates file space fragmentation; that is, a URL always occupies consecutive bytes within a file. This simplifies the mapping between URLs and the positions within files where they reside.

[5] BUDDY may also be used to reduce internal fragmentation and improve hit ratio by storing more data on a given disk. Current architecture trends suggest that disk block size should increase. This implies that small files, which occupy at least one disk block, in the future will probably occupy significantly more space than needed. On the contrary, when storing several URLs per file, as BUDDY does, internal fragmentation will be reduced, more data will fit into the disk, and higher hit rates will be possible.

To evaluate the performance advantages of BUDDY we compared it against three traditional approaches:

- SIMPLE: this approach stores each URL in a separate file. All files reside in the same directory.
- SQUID: this approach, used by the SQUID proxy server, creates a two-level directory structure. The first level contains 16 directories (named 0..F), while the second level contains 256 directories (named 00..FF) for each first-level directory. Files are written in the directories in a round robin manner: the first file is written at 0/00, the next at 0/01, ... then at 0/FF, then at 1/00, etc.
- MULTIPLE-DIRS: this approach creates one file for each URL. All files that correspond to URLs from the same server are stored in the same directory. Files that correspond to URLs from different servers are stored in different directories. All directories are in the same level.

Figure 4 shows the completion time of a stream of 300,000 file-system requests (URL-read, URL-write, URL-delete), which were generated by 398034 URL-get requests as a function of the management algorithm used. We see that SIMPLE has the worst performance, serving about 14 URL-get operations per second. SQUID performs better - it achieves 20 URL-get operations per second; independent published performance results also suggest that SQUID achieves 20-25 URL-get operations per second on a single-disk system [9]. MULTIPLE-DIRS performs a little better, achieving 23 URL-get operations per second.

Compared to SIMPLE, SQUID, and MULTIPLE-DIRS, BUDDY improves performance almost by an order of magnitude, since it achieves close to 133 URL-get operations per second. This is because BUDDY neither creates nor deletes files for most of the URLs it serves.

The careful reader however, will notice that SIMPLE, SQUID, and MULTIPLE-DIRS appear to be more robust than BUDDY in a case of system crash. If the system crashes, SIMPLE, SQUID, and MULTIPLE-DIRS will probably recover a large percentage of their metadata, while BUDDY will probably lose some portions of its metadata (i.e. where is each URL stored). We believe that this is not a significant problem for the following reasons:

- BUDDY can periodically (i.e. every few minutes) write its metadata information on safe storage, so that in the case of crash it will lose only the work of the last few minutes.

- Alternatively, BUDDY can store along with each URL, its name and size. 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 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.
- There exists a significant amount of recent work that speeds-up synchronous disk write-operations (and thus metadata updates) by using for example Non-volatile RAM [42], transactions [14], replication [30], or soft-updates [25].

In BUDDY, URLs that are larger than a threshold are stored in a separate file each - one URL per file. All other URLs are "buddied" together in appropriate files. The next experiment sets out to explore how large this threshold should be. Figure 5 plots the completion time of the BUDDY as a function of the threshold. We see that as the threshold increases, the completion time of BUDDY improves fast. This is because an increasing number of URLs are stored in the same file, eliminating a significant number of file create/delete operations. As the threshold increases above 10 (disk blocks), the completion time improves, but not as fast. When the threshold reaches 256 blocks (i.e. 128 Kbytes), we get (almost) the best performance. Our results suggest that 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.

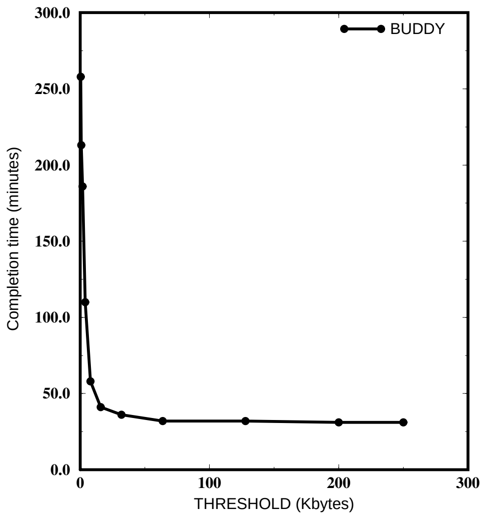

Figure 5: **Overhead of BUDDY as a function of the THRESHOLD.** The figure displays the cost of serving 400,000 file system operations as a function of the threshold used by BUDDY. The experiment suggests that URLs smaller then 128 Kbytes should be "buddied" together. URLs larger than 128 Kbytes can be safely given a file of their own (one URL per file) - they will not result in any noticeable overhead.

3.5 Optimizing Write throughput

Once we reduce the file management overhead we noticed that the next single largest source of overhead is due to disk latencies incurred by writing data scattered all over the disk. Although it reduces file management overhead, BUDDY makes no effort to layout data on disk in such as way as to improve write (and/or read) performance. Given that a web proxy's disk workload is write-dominated (as shown in figure 2), the performance of write operations can be improved if writes to the disk happen in a log-structured fashion. Thus, instead of writing new data in *some* free space on the disk, we continually *append* data to the disk until the disk runs out of space, in which case write operations continue from the beginning of the disk. This method has been widely used in log-structured file systems [5, 15, 28, 35]. In this paper we use a log-based approach in *user-space* management to see if the effectiveness of log-structured file system can be achieved by a user program (the web proxy) without the need of a specialized file system. 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 such slots.
- URL-delete operations mark the appropriate space on the file as free.
- 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 file runs out of space (i.e. they reach the end of file). In this case, new URL write operations continue from the beginning of the file writing on free slots, until they reach the end of file, etc.

STREAM has the potential of making long sequential write operations. The length of these sequential write operations depends on the distribution of the free space on the file, which in turn depends on the amount of scratch space that is available to the file. For example, if there is no scratch space, then there will always be only one free slot in the file, which will tend to move non-sequentially in the file, and STREAM will have little opportunity to make long sequential writes. The whole purpose behind STREAM (and log-structured file systems) is that disks should be operated at much less than 100% of their utilization, so that there is always enough free space on the disk. This free space will be used to write new files/data in long sequential write operations.

When we first evaluated the performance of STREAM we noticed that even when there was always free space 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 a *disk-read* and a *disk-write* operation. The reason is the following: if a process writes a small amount of data (e.g. the first block of a page) in a page that is not in the main memory cache, the operating system will *read* the page from the disk, make all updates in main memory, and then write the page to the disk. To reduce these small-write effects we developed a packetized version of STREAM: STREAM-PACKETIZER that works just like STREAM with the following exception:

- There exists a packetizer buffer that is one page long and aligned to a page boundary. URL-write operations are not being forwarded to the file system - instead they are being accumulated into a packetizer 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 for writing to the disk.

Figure 6 plots the performance of BUDDY, STREAM, and STREAM-PACKETIZER as a function of disk utilization. When disk utilization is high (around 95%), STREAM and STREAM-PACKETIZER perform comparable to BUDDY. This is because, at 95% utilization there do not exist long sequential portions of free space, and thus STREAM and STREAM-PACKETIZER can not perform long sequential write operations. On the contrary, when disk utilization is less than 72%, STREAM performs two times better than BUDDY, and STREAM-PACKETIZER performs 2.5 times better than BUDDY. Actually, STREAM-PACKETIZER manages to achieve more than 350 URL-get operations per second. To deliver their high performance, STREAM and STREAM-PACKETIZER need about 30% more disk space than the actual size of the URLs they need to store. Fortunately, the cost of disk space decreases rapidly (by a factor of two) every year [10]. Recent measurements suggest that most file systems are about half-full on the average [11], and

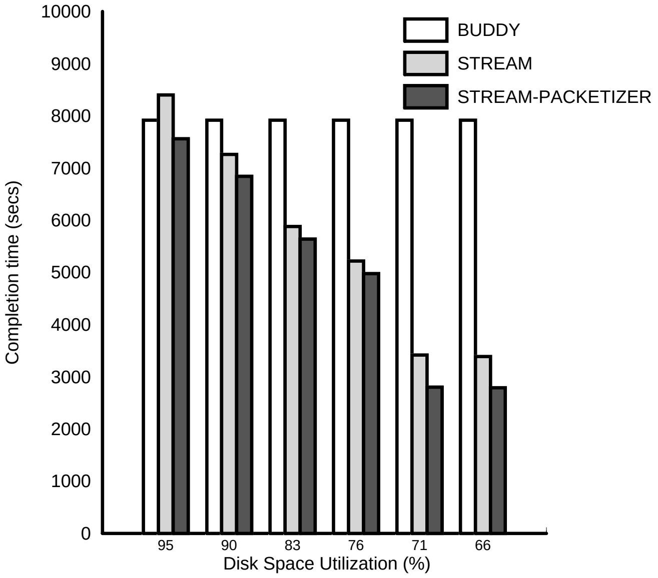

Figure 6: **Overhead of file management algorithms as a function of disk (space) utilization.** The figure displays the time it took to serve 1,000,000 file system operations as a function of disk utilization. The performance of STREAM and STREAM-PACKETIZER improves as disk utilization decreases. When disk utilization is around 70% both STREAM and STREAM-PACKETIZER outperform BUDDY by 2-3 times.

thus, log-structured approaches for file management may be more attractive than ever.

3.6 Improving Read Requests

Thanks to the STREAM and STREAM-PACKETIZER algorithms, URL-write operations suffer little (if any at all) seek and rotational overhead. However, 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 it must read data. To make matters worse, once the read operation is completed, the head must move back to continue streaming its data onto the disk. Thus, each read operation (which necessarily happens within a stream of write operations) induces *two* head movements. To reduce this overhead we have developed a LAZY-READ approach which is much like STREAM-PACKETIZER with the following difference:

> Once a URL *read* operation is issued, it is being sent into an intermediate buffer. When the buffer fills up with read requests it forwards them to the file system, sorted according to the position (in

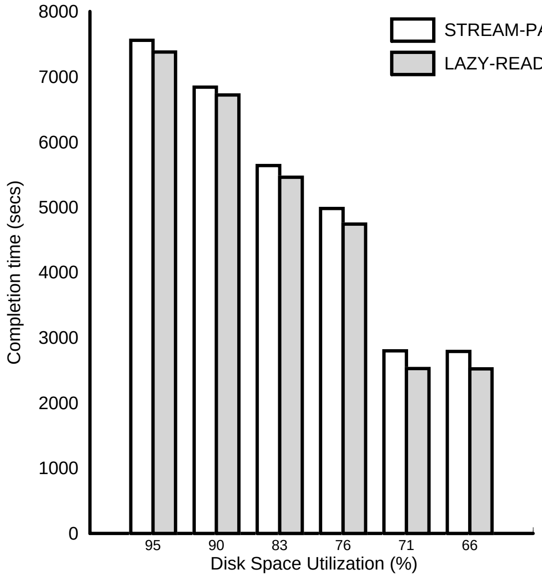

Figure 7: **Performance of LAZY-READS**. The figure displays the cost of serving 1,000,000 file system operations as a function of 2-Gbyte disk utilization. LAZY-READS gathers reads requests ten-at-a-time and issues them all at the same time to the disk reducing the disk head movements between the write stream and the data read. The figure shows that LAZY-READS improves the performance of STREAM-PACKETIZER by 10%.

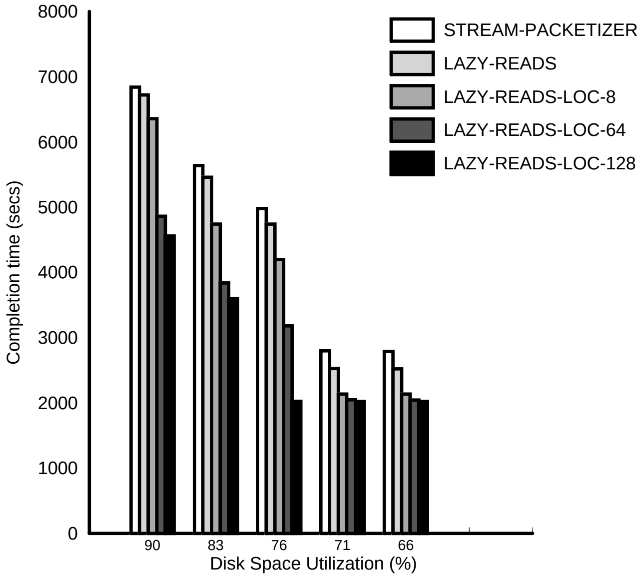

Figure 8: **Performance of LAZY-READS-LOC**. The figure displays the cost of serving 1,000,000 file system operations as a function of disk utilization. LAZY-READS-LOC attempts to put URLs from the same server in nearby disk locations by clustering them in locality buffers before sending them to the disk. We see that even as few as eight buffers improve performance over LAZY-READS.

the file) of the data they want to read.

Figure 7 shows that LAZY-READS improves the performance of STREAM-PACKETIZER by 10%. It is true that we expected a larger performance improvement. We traced the operating system actions and found that even if LAZY-READS sends read operations to the file system ten-at-a-time, the file system does not preserve this clustering and sent 3-5 clustered read operations to the disk in the average. Nevertheless, clustering read operations has potential and should be further explored. [6]

[6]The careful reader will notice however, that LAZY-READS may increase operation latency. 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 is several seconds long [2], LAZY-READS impose an unoticeable 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.

3.6.1 Preserving the Locality of the URL stream

The URL requests that arrive in a web proxy contain a significant amount of locality. For example, consider the case of an HTML page that has several embedded images. Every time a user requests that HTML page, (s)he will probably request all the embedded images as well. Thus, it may be worthwhile to store the HTML page and its embedded images in nearby disk locations so that future accesses to the HTML page and its embedded images will proceed at top speed. Unfortunately, current proxy servers tend to destroy such locality because they receive (and interleave) requests from several web clients. Thus, contiguous requests from a single web client may be received by the proxy server interleaved with tens of requests from other clients. Therefore, URLs that correspond to contiguous requests from a single client may be stored in the magnetic disk hundreds of Kbytes away from each other. To remedy this problem we have augmented the LAZY-READS policy with a number of locality buffers (LAZY-READS-LOC) that work as follows:

- There exist a set of locality buffers whose purpose is to accumulate URL-write operations that correspond to URLs from a single web server.

Algorithm	STREAM-PACK	STREAM-PACK +LAZY-READS	STREAM-PACK +LOC-128	STREAM-PACK +LAZY-READS +LOC-128
Time (minutes:sec) Improvement (over STREAM-PACK)	46:31	42:08 10%	36:16 28%	33:51 37.5%

Table 1: **Comparison of the improvement of LAZY-READS, and locality buffers on the STREAM-PACKETIZER algorithm.** We see that the single largest performance improvement (28%) comes from the use of locality buffers and the next improvement (10%) comes from LAZY-READS.

- When the proxy wants to store a URL fetched from some web server, it searches for a buffer that accumulates URLs from the same server and adds the data to the buffer. If no such buffer is found, one victim buffer is selected, its contents are written to the disk, and the new URL is written in the buffer.

This policy gathers URLs from the same server into the same locality buffer, so that URLs from the same server requested within a short time interval will probably be written in contiguous file locations. We have evaluated the performance of this policy (for 8-128 locality buffers) against the performance of LAZY-READS and STREAM-PACKETIZER. Figure 8 plots the results. We see that the existence of even eight locality buffers (LAZY-READS-LOC-8) improves performance over LAZY-READS significantly. The most spectacular improvements happen at medium to large disk utilization. For example, at 76% disk utilization LAZY-READS-LOC-128 performs 2.5 times better than LAZY-READS. In all cases, however, LAZY-READS-LOC-128 is at least 30% better than LAZY-READS. In the best case LAZY-READS-LOC achieves around 500 URL-get operations per second.

In our final experiment we will explore what is the contribution of each factor (read-clustering/LAZY-READS and locality buffers/LOC) to the performance of STREAM-PACKETIZER. Table 1 presents the completion time of policies STREAM-PACKETIZER, STREAM-PACKETIZER augmented with LAZY-READS, STREAM-PACKETIZER augmented with locality buffers (128 of them), and finally, STREAM-PACKETIZER augmented with both LAZY-READS and locality buffers at 71% disk utilization. It also shows the (percentage) improvement of every method on top of STREAM-PACKETIZER. We see that LAZY-READS improve 10% on STREAM-PACKETIZER, locality buffers improve 28% on STREAM-PACKETIZER, and both methods improve 37% on STREAM-PACKETIZER.

Summarizing, table 2 shows the (best) performance of the various algorithms studied.

Algorithm	Performance (operations per second)
SIMPLE	14
SQUID	20
MULTIPLE-DIRS	23
BUDDY	133
STREAM	295
STREAM-PACK	358
LAZY-READS	396
LAZY-READS-LOC	495

Table 2: **Comparative performance (in terms of URL-get operations per second) of various file space management algorithms.**

4 Summary-Conclusions

In this paper we study the disk I/O overhead of world-wide web proxy servers. Using a combination of experimental evaluation and simulation based on traces from busy web proxies we show that web proxies experience significant overheads due to disk I/O. We propose several file management methods (like BUDDY, STREAM, LAZY-READS, STREAM-PACKETIZER, and locality buffers) which reduce the disk management overhead by more than a factor of 25 overall (from SQUID to LAZY-READS-LOC). Based on our experiments we conclude:

- The single largest source of overhead in traditional web proxies is the file creation and file deletion overhead associated with storing each URL on a separate file. Storing several URLs per file improves performance by an order of magnitude.
- Disk accesses of web proxies are dominated by write requests. Streaming these write operations to disk (much like log-structured file systems do) improves performance by a factor of 2-3.
- Web clients display a locality of reference in their ac-

cesses. Web proxies tend to destroy it by interleaving requests from several clients. Preserving this locality of reference results in better layout of URLs on the disk, which improves performance by 30%-150%.

- User-level file management policies improve performance (over traditional web proxies like SQUID) by a factor of 25 overall, leaving little space 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 [10], methods that reduce disk head movements and stream data to disk will result in increasingly larger performance improvements.

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), and in part by EPET II project "E-Commerce" funded through the General Secretariat for Research and Development. 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 (our shepherd) provided useful comments in the final 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. *Journal of Computer Networks and ISDN Systems*, 30:2179–2192, 1998.

[3] 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.

[4] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *roceedings of CIKM'95: The Fourth ACM International Conference on Information and Knowledge Management*, November 1995.

[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 USENIX Symposium on Internet Technologies and Systems*, 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] G. Chisholm. Squid Performance as a Factor of the Number of Disks Utilised. http://www.dnepr.net/Squid/Benchmarking/Number-of-Disks/.

[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] 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.

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

[14] R. Hagmann. Reimplementing the Cedar File System using Logging and Group Commit. In *Proc. 11-th Symposium on Operating Systems Principles*, pages 155–172, 1987.

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

[16] 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.

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

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

[19] T.M. Kroeger, D.D.E. Long, and J.C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, pages 13–22, 1997.

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

[21] 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.

[22] 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.

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

[24] E.P. Markatos and C. Chronaki. A Top-10 Approach to Prefetching on the Web. In *Proceedings of the INET 98 Conference*, 1998.

[25] M.K. McKusick and G.R. Ganger. Soft Updates: A Technique for Eliminating Most SynchronousWrites in the Fast FileSystem. In *Proc. of the 1999 Usenix Technical Conference - Freenix Track*, pages 1–17, 1999.

[26] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of the 1996 Usenix Technical Conference*, pages 279–294, January 1996.

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

[28] 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.

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

[30] Athanasios Papathanasiou and Evangelos P. Markatos. Lightweight Transactions on Networks of Workstations. In *Proc. 18-th Int. Conf. on Distr. Comp. Syst.*, pages 544–553, 1998.

[31] 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.

[32] 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.

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

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

[35] 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.

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

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

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

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

[40] 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.

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

[42] Michael Wu and Willy Zwaenepoel. eNVy: a Non-Volatile Main Memory Storage System. In *Proc. of the 6-th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 86–97, 1994.