Dámelo! An Explicitly Co-locating Web Cache File System

by

Jonathan T. Ledlie

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science
(Computer Science)

at the
UNIVERSITY OF WISCONSIN - MADISON

2000

Abstract

How to best relate cached web objects is a complex and on-going problem and always depends on the workload. By locating objects which are likely to be referenced together adjacently on disk and then importing them into memory as a group, large reductions in user-perceived read latency are possible. Instead of discovering new policies for relating objects, we have built a specialized file system called Dámelo *which leverages its data's cache-only nature for speed while leaving the determination of how to relate objects as a flexible policy external to the system. By providing a simple but effective interface for co-location, Dámelo provides much higher performance for its specialized web object workload than a standard Unix file system. The purpose of this thesis is to design, implement, and test the Dámelo file system with microbenchmarks and real workloads and to build a multithreaded networking layer to propel web objects through it.*

1 Introduction

Cachable web objects embody certain unique properties which enable — and demand — much higher performance than when they are stored in a standard UNIX file system as part of a caching proxy. While various elegant methods have attempted to keep frequently accessed files in-memory, one study has portrayed that approximately 90% of all hits for files at a proxy come from disk and that disk delays contribute 30% toward total response time: in memory speedups can only go so far [18]. As network speeds advance and disk access rates stagnate, this disk delay percentage can only increase. Two other studies have established that arranging files by size and domain name can improve throughput by as much as 25 times over a naive allocation within a simple directory hierarchy [11][12]. These systems also leverage the unchanging size and lack of ownership inherent in cacheable web objects by storing some "related" objects in the same file. Fetching one of these objects leads to a prefetch for all. One system further improves performance by buffering writes to make for more sequential access, often at the expense of reads. Both systems use a standard file system.

How to best relate cached web objects is a complex and on-going problem and always depends on the workload. Instead of coming up with more policies which work well under some workloads, we have designed, implemented, and tested a specialized file system called *Dámelo* (*give it to me* in Spanish) which leverages its data's cache-only nature for speed while leaving the determination of how to relate files as a flexible policy external to the system. That the objects already have a "backup" is true by definition and enables us to loosen the strict consistency semantics that a standard file system enforces. By keeping metadata in memory and checkpointing based on how much we are willing to lose with a crash, Dámelo achieves a balance between robustness and speed.

Dámelo replaces the standard mechanism of a file system with one which both exposes relations between objects and allows for user-defined robustness. Dámelo has two tuneable "knobs" which reflect the twin foci of prior research:

1. which objects are related to one another
2. which sets of objects to keep "hot" or in memory

Objects' interrelationships can be based on when the requests occured, if they came from the same client, if they were for the same server, how often an object is referenced, or some other metric or any combination of these. Storing related files together on disk can be simulated by placing them in the same directory, as earlier research has done, but this attempt at co-location works more and more poorly over time as inode references become scattered over distant cylinders. In Dámelo, related objects go into the same *group* which then directly translates into on-disk adjacency, even as the disk fills up.

The second major policy which caches must determine is which objects to keep resident in memory. This form of buffer management is less akin to a file system and more to a database. Like object interrelations, determining which objects are hot is a complex problem. After grouping related objects, a proxy can *hint* to Dámelo which to keep in memory. The proxy, making these determinations at run-time, decides to *love* or *hate* groups based on their anticipated usage. Loved groups stay hot; hated ones percolate to disk.

Dámelo's purpose is to function as a fast file cache while turning previously implicit, rigid policies into explicit, malliable mechanisms. Section 2 discusses previous policies for relating objects and the origins of Dámelo's underpinnings, like checkpointing and in-memory metadata. Section 3 portrays how a researcher would use the thread-safe API. Section 4 looks at how Dámelo works beneath its interface. Section 5 examines results from microbenchmarks and a sample proxy. Section 6 looks at future modifications and Section 7 summarizes the project and its results.

2 Related Work

A web proxy cache's goal is to reduce the latency between clients and servers. The primary method for achieving this aim is to retain a valid copy of a subset of the data clients have asked for before, in anticipation that the at least one of the clients will request the same data in the future. After some (possibly zero) length of time after the data passes through the proxy on its route down from the server to the client, the data expires, causing the next request for the expired data to miss in the cache and the server to generate a fresh copy with a new expiration date. Projects to first research this method for reducing Internet latency were Harvest[2], its sucessor Squid[21], and CERN[10]. Figure 1 shows how Dámelo fits between a proxy (like Harvest, Squid, or CERN) and the disks where it would store the bulk of its data.

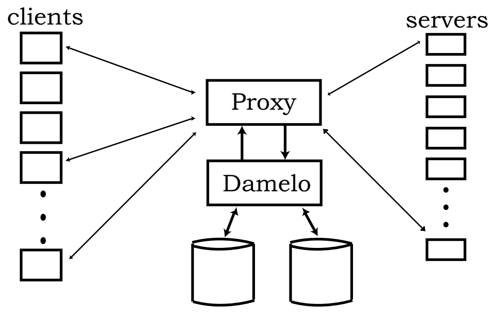


Figure 1: Where Dámelo fits in

One key concept originating in these first web caches is that data can simply expire in the cache, causing a miss and then a new fetch from the server with no harm except for a client-perceived delay. As long as Dámelo can detect inconsistencies in cached copies and then *expire* bad data before it is sent to the client, it can be far more aggresive in storing data than either a typical database or file system. This fact that the data we are storing is entirely replaceable is one of the three invariants which differentiate Dámelo from a typical file system.

The two other pieces of information that Dámelo does not need to store are ownership and execution permissions. Once in transit, Internet data is anonymous. Similarly, any execution priviledges are encapsulated in the data itself, e.g. in the header of a multimedia file. Other file system work has shown that by taking advantage of invariants, a much faster solution is often possible[16].

More recently cacheable data has begun to include dynamically generated content, the results of CGI queries, for example [9]. It could store this data but Dámelo would need to sit beneath the active caching intelligence, just as it currently resides under a proxy. An active cache's datum might be much larger than the typical static web object which was tested in Section 5. For example, an active cache might want to refer to a query table as a single object, in order to always transfer it to and from memory as one unit. If a cache were to handle both types of data concurrently, it would probably benefit from multiple page sizes. This will be discussed in the section on future work.

2.1 Relating Web Objects

Much research has gone into how to best interrelate web objects and how to exploit their unique characteristics. [11] lists most of their significant attributes:

- Objects are always read in their entirety. We add the observation that they are never modified locally.
- 74% of web objects are $\leq$ 8kb and 99% of transfers are smaller than 64kb [18].
- Web objects' popularity follows a Zipf-like distribution, which means that some files are much more popular than others, although this popularity does evolve over time [3].
- Embedded objects (like inline gifs) and intra-domain-name references produce high data and meta-data locality.
- Objects are redundant; they are by definition "backed up."

On redundancy, the authors note: "it is acceptable to never actually store web objects to disk or to periodically store all objects to disk in the event of a server crash." We concur and independently explored this idea in [14]. In addition to these attributes, [11] lists a low ratio of disk reads to writes because "every miss involves a read of the cache meta-data [and] a write of the meta-data." If the redundancy of the data affords us the freedom to keep much of it and its meta-data in memory, however, these I/O operations disappear.

[11] and [12] independently researched methods for preserving locality. By locating objects which are frequently accessed together nearby, they hope to prefetch some objects and to have their meta-data cached to reduce I/O. In [11] the authors modified Squid to map domain names to particular directories, but found that "a single directory may store objects from a popular server. This can lead to directories with many entries which results in a directory spanning multiple blocks." As a typical Fast File System (FFS) [8] like Linux's ext2 fills up and becomes fragmented, however, these multiple blocks will no longer be in the same cylinder and maybe not in the same cylinder group. [12] takes the domain name mapping one step further and places small objects which hash to the same directory into the same file. They call these objects "buddies." Their idea is to reduce the meta-data overhead and fragmentation caused by many small files. As noted above, however, this frequently accessed meta-data lends itself to remaining in memory, alleviating this problem entirely. Dámelo also solves the fragmentation

problem by compacting its buffers, putting all free space at the end of each.

[11] takes its related web objects and places them in memory-mapped files. The authors then align these objects on page boundries to circumvent undesired paging. Dámelo takes this workaround and makes it explicit: objects go into pages which are aligned but which are swapped out when *explicitly* told to do so, not when a general purpose operating system algorithm believes is correct.

Similarly, [12] buffers its *buddies* into "streams," which it keeps in memory for a period long enough to make the write to disk worthwhile. When full, each *stream* is written out atomically, much like a Dámelo page after it has been *hated.* This uninterupted streaming approach, combined with locality, achieves 495 URL-get-operations per second vs. 20 for Squid's naive scheme.

An alternative approach is to design an operating system especially for caching. The CacheOS improves response time with "object pipelining," which opens as many simultaneous TCP connections as the origin server will allow and retrieves these object in parallel [4]. This algoritm appears to be a refinement on the multithreaded approach used in subsection 5.4, in that it parses each HTML document on its way back to the client and requests inline objects automatically. Still, this algorithm clearly could be incorporated into a proxy running on a conventional operating system.

The difficultly with these solutions is that they either rest on a standard file system or are operating systems unto themselves, suffering from portability problems. When incorporated into a regular file system, [11] and [12] are not taking advantage of the invariants their data afford. Thus, they study the best policies for how to relate one object with another but still perform the actual storage with a mechanism which is far more flexible than they need. With an FFS-based file system, they lose opportunities for optimization. Here, in Dámelo, we have designed a user-level file system which works with the same commonly used operating systems as their research, but is significantly faster because it is not constrained by the semantics of general-purpose irreplacable data.

2.2 Constituent Mechanisms

The primary source for Dámelo's explicit grouping idea is [7]. The authors "aggressively pursue adjacency of small objects rather than just locality." Like Dámelo, they espouse the idea of large blocks: "a 64kb access takes less than twice as long as an 8kb access." Because they offer conventional file system integrity as part of their Co-locating Fast File System (C-FFS), they keep their inodes on disk, but embedded within the files themselves. Grouping and embedding inodes make up the two halves of their solution.

While the idea of a specialized, but portable, file system for caching web objects is a new one, Dámelo internal mechanisms include ideas from many other research system sources. The origin for the *streams* above and here for appending additions to new groups in memory is the Log-Structured File System [17]. Flushing Dámelo's groups' metadata in the background and at distinct times comes from [13] and [1]. The concept of less-than-ACID semantics is discussed more fully in [6]. Attempting to minimize in-memory copying, in particular to a network port, was utilized in [20]'s zero-copy. Placing large objects toward the outside of the disk platter is discussed more fully in [15]. Dámelo's buffer manager and internal interfaces grew out of the Minirel project [5].

The speedups possible through in-memory metadata were explored by the author and Matthew McCormick in [14].

3 Interface

Dámelo is designed to work with a multithreaded server, be it a web proxy or some other source of cache-only data. Its designers felt that making the library thread-safe was necessary because each file request has the potential to block for a relatively long time, during which other threads could be sending responses out to clients, forwarding requests to servers, and, in particular, accessing data in Dámelo's memory pool.

In addition to allowing multiple threads to create, read and delete objects concurrently, we designed Dámelo to minimize in-memory copying. To this end, each reading thread is given a pointer directly to the spot in the memory pool where its object is located — it is not given a copy of the data. Similarly, as will be discussed in the Internals section, when pages of the memory pool are written to disk, they are not copied elsewhere first (as would happen with a kernel-managed block device). This reduction in memory creation, deletion, and copying also allows for a more accurate sum of total memory usage — a problem with Squid, for example — which in turn means that a cache administrator can allocate more physical memory to the system without resulting in undesired swapping.

Dámelo links as a library and header file into an application. The main thread of the application calls Dámelo's constructor.

```
Damelo (char *raw disk device name,
        int number of groups,
        int memory pool size,
        int &status)

~ Damelo ();
```

The raw disk partition should have already been created and given *rw* permissions for the user id the application runs under (see the *raw* command under Linux; under Solaris these unbuffered partitions should already exist). The number of groups signifies how many groups Dámelo should anticipate handling: this controls how the disk is divided and sets up data structures for each group. Group numbers range from 1 to n and are allocated alternating from the center of the disk. Visually, this looks like Figure 2 (assuming k is odd).

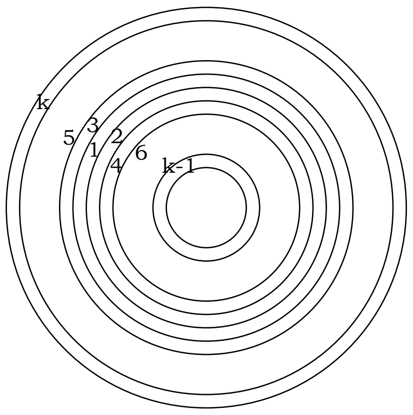


Figure 2: Disk layout

Because requests are handled using a two-way elevator algorithm, groups with low numbers will have lower response times because the disk head will pass over them more often. Because the outer tracks can achieve greater bandwidth due to their higher rotational speed, large objects should be placed in these groups [15]. This stage is still under development; please see Future Work for more detail.

The memory pool size parameter should be some multiple of the page size (because there is no point in having room for $\frac{1}{2}$ a page, for example). It is given to the constructor in megabytes. In our tests, we used pool sizes of 128M and 256M — a minimal amount of memory typically available on most machines. Pages and frames have the same size and this size is currently compiled into the library. Pages refer to data which is moved as an atomic unit and frames to the entities which contain them; pages migrate between memory frames and disk frames.

The status parameter for the constructor is passed by reference and allows, like the other return codes, any errors to be deciphered. Like the return values from the other functions, any non-zero status signifies an error and can be sent to Dámelo's *perror()* for a text explanation.

Dámelo's destructor forces all pending deletes, flushes all dirty buffers to disk, checkpoints the groups, and frees the memory pool.

```
int create (char *file name, char *data,
            int size, int group number)
```

create() takes a file of a given size and copies its data into the specified group. The assumption here is that objects are write-once, read-many. In a web proxy context (and in the one built for the Experiments section) files are assumed to have been sent to the requesting client before the proxy *create()*s the file in Dámelo. The reason the proxy would forward the response to the client first is that then this *create()* operation is taken out of the client's latency. The proxy always has the data in its entirety before calling *create()*.

Choosing in which group to place an object is exactly the complex question previous research has attacked and which Dámelo is designed to make simpler. How objects interrelate has many criteria. A simple scheme would hash domain names to the same group. A more complex one would try to pick out which objects are more commonly accessed than others and put those into the lower numbered groups. Any statistical information on a group or object could be kept in a header as part of the object, in another object in the group, or all this information could be placed together in its own group, or any combination of these. Keeping this meta-information in the objects themselves is not the best solution, however, because then a page might need to be saved to disk, even if its actual data had not been modified.

If the proxy anticipates creating many objects in the same group, it can *love()* the group, as will be discussed more below. For example, a scheme with hysteresis would automatically love a group if two objects were created there in a row and then *hate()* it once actions on this group ceased.

To simplify Dámelo's internal mechanisms, every object must fit into one frame. Because it is built on a regular operating system, larger objects can be cached in its regular file system, in another mechanism, or not at all. We do not see this as a drawback because the main penalties with a standard file system are seen with small objects, where seek times dominate. Also many of these large objects are multimedia files, which would probably use a disk streaming mechanism instead of Dámelo's which always reads objects in their entirety.

```
int lookup (char *file name,
            int &group number)
```

lookup() takes a filename and either returns 0 and the file's group number by reference or an error code signifying the file was not found. The purpose of this function is so that a program using Dámelo neither needs to remember which files it has created nor which groups they were put in. Thus, if the group assignment mechanism evolves over time (e.g. a server name formerly mapped to a low priority group but now maps to a high one), old files can still be found. Generally, *lookup()* is called directly before *read()* or *remove()* in order to find the object's group number.

```
int read (char *file name,
          int group number,
          char *&data, int &size)
```

```
int release (char *file name,
             int group number)
```

read() and *release()* are used as a pair to acquire a pointer into the memory pool where Dámelo has positioned the requested object and then to signify that the proxy's thread has finished reading the data and its space can be used for another page. Because we want to avoid copying data and because how long a thread will take to send this object out to the client is unknown, Dámelo provides a shared lock on the page where this object is located. *release()* releases this lock. Other threads can *read()* and *remove()* objects from this page while this shared lock is in place.

```
int remove (char *file name,
            int group number)
```

remove() prevents future access to the object and frees up space in its group. As will be explained in detail in the Internals section, deletes are buffered so that they only affect in-memory pages and data structures. Like *create()*, *remove()* does not return a locked page so no second function call is required.

```
int love (int group number)
```

```
int hate (int group number)
```

As stated in the discussion on create(), anticipated use of a group should be preceeded by a *love()* of the group and anticipated disuse by a *hate()*. These translate into LRU or MRU buffer pool replacement, respectively. If the proxy's group management scheme does not choose to modify any group's loved or hated state ever during run time, all groups should be initially set *loved*; otherwise, no matter how many buffers the pool is allocated, only one would ever get used!

```
setCheckpoint (int group #, int time)
```

Dámelo achieves much of the speedup seen in the experimental results by keeping file metadata in memory. Thus, there is no inode to first reference before performing the data-fetching I/O. To limit the amount of data lost in case of a crash, it can periodically checkpoint this metadata. This level of robustness can be modified during runtime and at the granularity of a group: groups to which there is little current access could be selected for checkpointing and, if no checkpointing is desired, it can be switched off entirely. Because the checkpoint operations are at a group level, they do not bring the system to a halt. The *time* parameter signifies the number of seconds between checkpoints. If set to zero, checkpointing is turned off.

```
void perror (char *s, int error)
```

Like the standard C library call, Dámelo's *perror()* takes a string which is output to *stderr* before the string which describes the error condition.

4 Internals

Figure 3 portrays Dámelo's main data structures and lists the main steps in a create request, enumerated (a - m). It also shows two read requests concurrently asking for objects from the fifth buffer. The figure divides the system into five main components, which represent the stages a thread traverses from the network at the top to the disk at the bottom.

The initial Dámelo object, whose constructor was discussed in the previous section, creates n groups, a buffer manager, and forks a new disk thread.

Each group object contains a list of its pages are in memory and how much free space is on each of its pages (including those on disk). Using this information, the group can try to allocate a new web object on a page currently in memory. During its constructor, each group also creates a *file* object. *Files* encapsulate the disk operations of each *group* while the *group* objects themselves handle in-memory operations. Each *group* also contains a large in-memory hash table recording the logical location (a ⟨ page,slot ⟩ pair) of each of its files. In order to reduce memory usage, each web file name (a URL) is converted to its 16-byte MD5 equivalent before insertion into the file table. Because it stays in memory, this file name length is significant when it is multiplied by millions of entries.

The buffer manager is perhaps Dámelo's most interesting and complex member. Frames are always in one of three states: unused and on the free list, shared by one or more threads, or used exclusively by one thread. Deletes are buffered so that they only occur when a page has already been exclusively locked due to a pending create or read. If a requested page is not in the pool, both create and read acquire an exclusive lock on an unused frame (frames 1 and 3 in Figure 3). If this frame already contains a valid, dirty page with pending deletes, the deletes and subsequent compaction are performed just before the victim page is sent to disk. The disk then reads in the requested page and, still with the exlusive lock, deletes objects from this page (the one which actually generated the request). The benefits of buffered deletes are twofold:

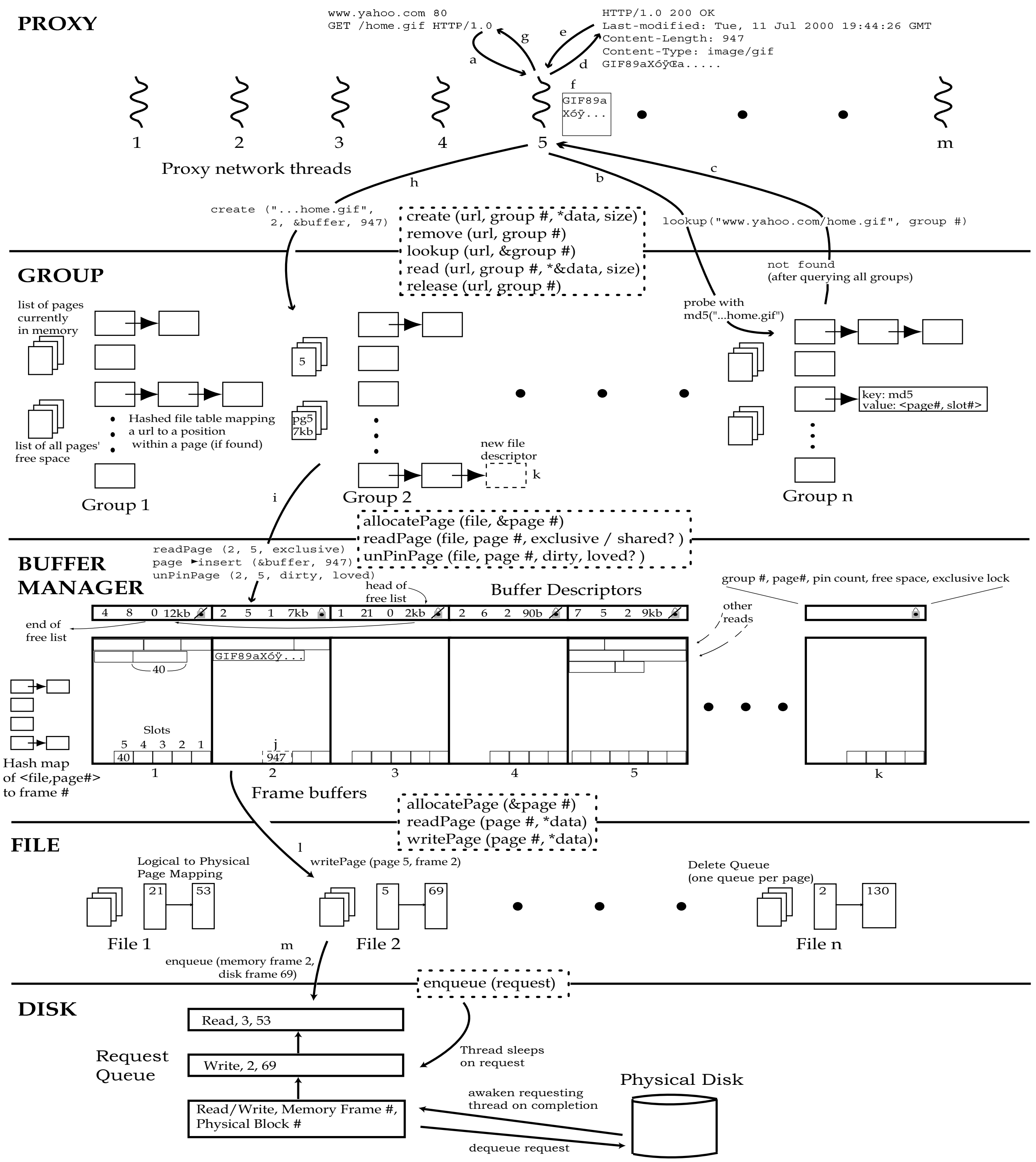
1. a delete request is changed from two I/O operations (read-modify-write) to zero.

2. they allow requests for deletions for the same page to be sorted into their most efficient order.

This order is the one which requires the least memory copying during compaction. In the case where every object in a page is being deleted, if objects are deleted front to back, every object must be shifted forward during each deletion. By reordering them back to front, no memory copying is required at all.

Files map *groups'* logical page numbers into the *disk*'s physical frame offsets. When the buffer manager needs to read or write a page, it enters that *group*'s corresponding *file* to make this translation and to generate a request for the *disk*. The thread which has caused this action then sleeps within the *file* object until the request is complete.

The *disk* thread waits for *files* to enqueue requests. After handling each request, it signals the requesting thread and looks for any new requests on its request queue. If there are none, the *disk* goes to sleep. When a thread positions a request in the request queue, it keeps the requests sorted by physical frame number, with the first entry the one nearest the current position of the disk head.


```
www.yahoo.com 80
GET /home.gif HTTP/1.0
```

```
HTTP/1.0 200 OK
Last-modified: Tue, 11 Jul 2000 19:44:26 GMT
Content-Length: 947
Content-Type: image/gif
GIF89aXőÿŒa.....
```

```
create ("...home.gif",
        2, &buffer, 947)
```

```
lookup("www.yahoo.com/home.gif", group #)
```

```
readPage (2, 5, exclusive)
page ►insert (&buffer, 947)
unPinPage (2, 5, dirty, loved)
```

PROXY
a
g
e
d
f
GIF89a Xőÿ...
1
2
3
4
5
m
Proxy network threads
h
b
c
create (url, group #, *data, size)
remove (url, group #)
lookup (url, &group #)
read (url, group #, *&data, size)
release (url, group #)
GROUP
not found
(after querying all groups)
probe with md5("...home.gif")
list of pages currently in memory
5
pg5 7kb
key: md5
value: <page#, slot#>
Hashed file table mapping a url to a position within a page (if found)
list of all pages' free space
new file descriptor
k
Group 1
Group 2
Group n
i
allocatePage (file, &page #)
readPage (file, page #, exclusive / shared?)
unPinPage (file, page #, dirty, loved?)
BUFFER MANAGER
head of free list
Buffer Descriptors
group #, page#, pin count, free space, exclusive lock
4 8 0 12kb
2 5 1 7kb
1 21 0 2kb
2 6 2 90b
7 5 2 9kb
end of free list
other reads
40
GIF89aXőÿ...
Slots
5 4 3 2 1
40
j
947
Hash map of <file,page#> to frame #
1
2
3
4
5
k
Frame buffers
allocatePage (&page #)
readPage (page #, *data)
writePage (page #, *data)
FILE
l
Logical to Physical Page Mapping
writePage (page 5, frame 2)
Delete Queue
(one queue per page)
21
53
5
69
2
130
File 1
m
File 2
File n
enqueue (memory frame 2, disk frame 69)
enqueue (request)
DISK
Read, 3, 53
Write, 2, 69
Read/Write, Memory Frame #, Physical Block #
Request Queue
Thread sleeps on request
Physical Disk
awaken requesting thread on completion
dequeue request

Figure 3: Dámelo's Internals

Please refer to the Appendix for the enumeration of the create steps from Figure 3 and for a code fragment.

5 Experiments

5.1 Microbenchmarks

We designed several microbenchmarks to test Dámelo under a web-like workload and then ran one configuration on a commonly used web proxy simulator. The experiments reflect our contention that read latency matters most, because it cannot be removed from the user's path. Creates are streamed into buffers much like a Log File System and, with a multithreaded server, they can be entirely masked by the actions of other threads. Deletes are entirely in memory, but can potentially affect reads as discussed in Future Work. Reads are the focus of these experiments.

Each microbenchmark consists of three distinct stages: *create*, *read*, and *delete*. For each stage, t threads perform $\frac{n}{t}$ operations, where n is the number of objects. All threads finish before any proceed to the next stage. Before the experiment begins, each object is assigned a group in the Dámelo version, or a directory in the standard file system. These standard file system directories are set up as a two-tiered namespace of [a-z]/[a-z]. This mirrors Squid's directory usage.

To quantify how Dámelo improves read performance while keeping create and delete performance good, we have designed five web object orderings. Thus, each microbenchmarks is a $\langle$ stage, object ordering $\rangle$ pair, giving a total of fifteen microbenchmarks.

sequential all three operations are performed on the set of objects in the same order, and in directory or group order, [a-z] or [1-n] respectively.

random all f files are created, but are done so in a random order (e.g. a create for group 7 could preceed one for group 3). f number of reads follow, but each read could be for any file, with equal probability. Finally, all f files are deleted, in a different random order than they were created.

zipf 1:8 Here and for the other zipfs, creates and deletes are as in random. $\frac{1}{8}$ of the objects receive $\frac{7}{8}$ of the requests and the unpopular $\frac{7}{8}$ of the objects receive only $\frac{1}{8}$ of the requests. Among each fraction, the requests are randomly distributed.

zipf 1:4 Like Zipf1 : 8 except the ratio of popular files to unpopular files has been doubled.

zipf 3:8 Again like Zipf1 : 2 except almost half of the files have been marked "popular."

[18] and others have found that web objects follow a Zipf-like popularity distribution, where some set of objects are requested more frequently than others. The zipf experiments approximate this distibution. The ratio of popular files to unpopular can be thought of as either the actual distribution of requests or the accuracy of the proxy's object interrelation policy. In each of them popular objects come from the same groups (or directories in ext2) and these groups have been *loved* so they remain in memory. In sequential and random all of the groups are *loved*.

In between moving from the *create* to the *read* stage for each microbenchmark, Dámelo's buffer pool is flushed; no reads are hits left over from the *create* stage. A timestamp is recorded for each operation; the graphs are a sampling of these timestamps for easier readability. The average and maximum latency charts are gathered from all timestamps, not a sampling. The hit rates are the ratio of the number of times objects on a page are requested to the total number of objects on that page. In sequential read, all objects are accessed exactly once, making the hits-to-object count ratio always one. In random and zipf1 : n tests, the same objects can be accessed more than once, so the hit rate varies.

5.2 Experimental Setup

The experiments were conducted on a dual-processor 500 MHz Pentium III running Red Hat Linux version 6.2 with a stock 2.2.16 kernel. The machine has five IBM Model 9LZX SCSI disk drives of 9.1 Gigabytes each, and can sustain a throughput of between 180 and 240 Mbits/sec. It has 1G of RAM. All logging was done to separate disks on separate SCSI controllers. Except for the raw disk, the remainder were freshly formated with Linux's standard ext2 filesystem, with either 1kb, 2kb, or 4kb block sizes. 4kb is ext2's maximum block size. All of the microbenchmarks shown used 131072 objects of a random size of up to 16kb each: each test's objects used $8k\, bytes\, on\, average \times 128k\, objects \approx 1G$. Dámelo used 16 groups, 16 threads, a 128M buffer pool, and 1.5G of the disk. The buffer pool and disk were divided into frame sizes of 16kb, 32kb, 64kb, 128kb, and 256kb. At 128M, the buffer pool can hold $\frac{1}{8}$ of the total test size. Ext2 had available to it all of the 9.1G disk. It ran with 4 threads; experiments showed that ext2 performed slightly better with this fewer number of threads. Other tests, like with a larger buffer pool, showed similar results and they were omitted for clarity.

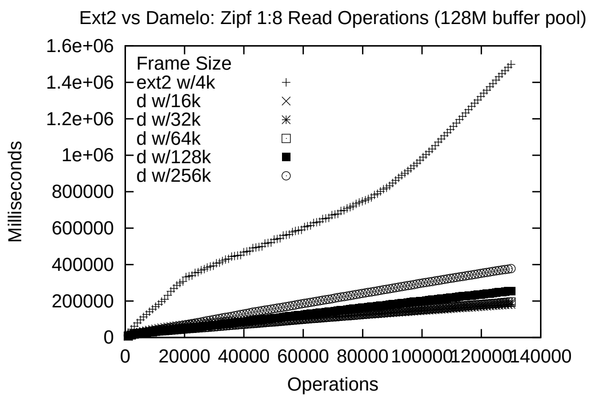


Figure 5: Ext2 vs Damelo: Zipf 1:8 Read

5.3 Microbenchmark Results

The results from the five microbenchmark tests are shown in Figures 8 to 25 in the Appendix. Dámelo's sequential performance is consistently better than ext2's and Figures 11 and 12 portray the benefit of larger transfer units, especially when all of each unit is utilized. The sequential read peaks at an average latency of about half a millisecond for the 256kb frames, and even larger frames would perform even better (Figure 6). Even with 16kb frames, Dámelo is five times faster for this test.

As would be expected, random read performance suffers with larger frame sizes in Figure 18. Even in the worst case, however, the average read latency is still on par with ext2. The sub-64kb frames all have clearly better performance on this metric.

With Dámelo, random creations (Figure 17) do not penalize the time per operation nearly as much as for ext2 (Figure 14), because Dámelo's objects append to each group's stream just the same as in sequential, whereas ext2 must seek to each direc-

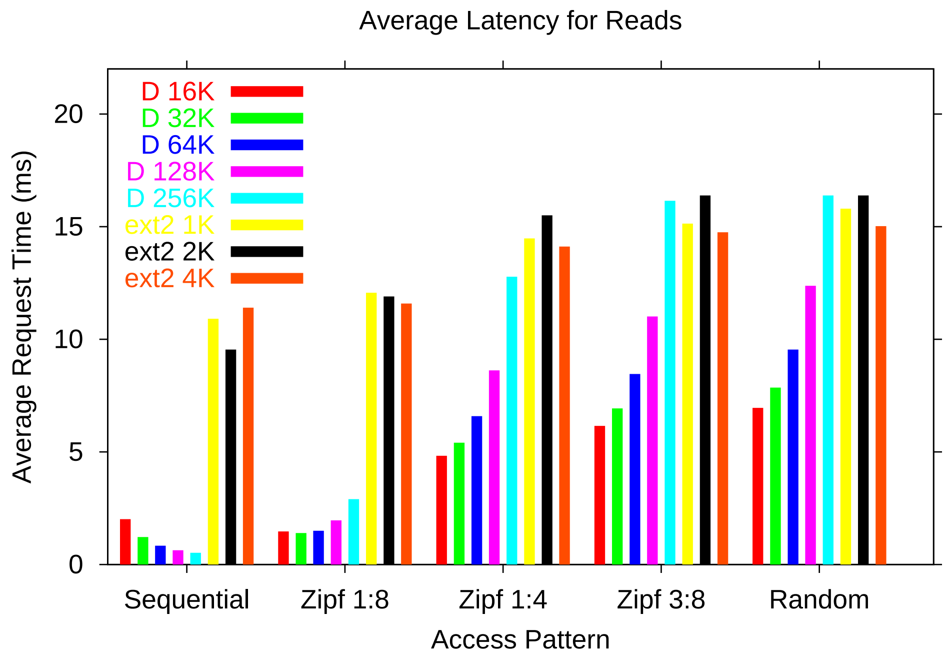


Figure 4: Average Read Latency

tory. With its delayed deletions, Dámelo's delete performance is one order of magnitude faster than ext2 (Figures 10 and 13). This speed is especially significant in a caching proxy context where deletions, even when handed off to a separate process, are often a great burden on the system [22].

The zipf experiments convey the power of Dámelo's two proxy-controlled knobs, the relation and "hit" mechanisms. When a proxy's relation algorithm has successfully placed popular objects into the same groups and *loved* them, we see a huge performance gain over ext2. Figure 4 and 5 portray a sixfold speedup over ext2 for zipf1 : 8 and between two and three times for zipf1 : 4. These two workloads benefit from good hit rates for the 32kb and 64kb frames (Figure 6).

The experiments show the tradeoff between large frames and hit rates. Clearly if only 8kb out of 256kb is used per I/O, like in Random, we get poor performance. With 16kb buffers, we are guaranteed a hit rate of at least 50% on average (because of the ≈8kb objects). A happy medium seems to emerge as the ability to discern the more popular objects becomes more accurate and the frame size increases. The beginnings of this trend are with 32kb having the best latency for zipf1 : 8. These results show that an accurate grouping mechanism leads to lower read latency.

5.4 Web Proxy Simulation

Due to time constraints, we were not able to explore the web proxy simulator as thoroughly as we would have liked. Still, we were able to set up and

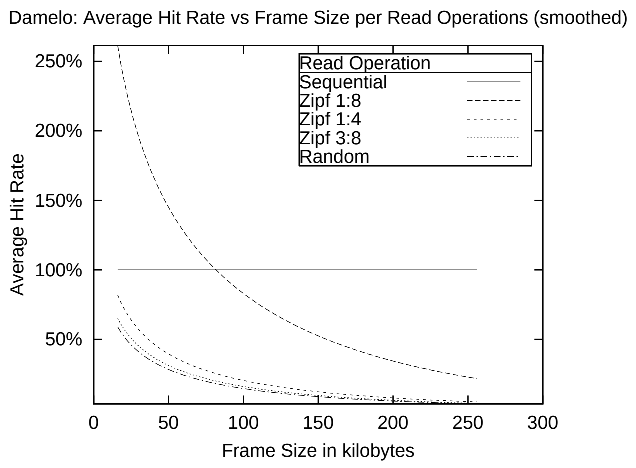


Figure 6: Hit Rate over varying Page Sizes

Proxy	Avg Requests/sec
mt-ext2	363
Squid	545
mt-Dámelo	617

Figure 7: Web Proxy Simulation Results

run some preliminary tests with Web Polygraph [19].

Web Polygraph's environment has c virtual clients and s virtual servers utilizing one or more proxies in between. In this experiment, we had one of each. We used three different proxies: *Squid*, a multithreaded proxy backed by ext2 (*mt-ext2*), and a multithreaded proxy backed by Dámelo (*mt-Dámelo*). The tests were run on the same hardware as above with the exception that each unit (the client, the server, and the proxy) was on a different machine. *mt-Dámelo* mapped objects to groups randomly. Both Squid and *mt-ext2*'s directories were cleaned before each test. The tests results show the average throughput (requests per second) as measured by the client in responses. The test has 80% of the data be cacheable, based on the header, and 55% of the data is revisited. The web object sizes are exponentially distibuted over a 13kb maximum. Dámelo was given 128M for its buffer pool and used 64kb frames and 16 groups. Each test was run for five minutes.

Although the sample proxy we built for this project was fairly primative and, therefore, could not cache much content as Squid could, it gets about a 13% improvement. The performance gain of 69% of the simple proxy using Dámelo vs. the one with ext2 (which used a directory structure like Squid's) suggests that with an better proxy the speedup of *mt-Dámelo* over *Squid* would be magnified.

6 Future Work

Beyond a more precise equation for determining the best page size based on the average object size and on the popularity of certain requests, there are several other items this project has left for future work.

One difficulty with the current implemenation of deletions is that it conflicts with our focus on read latency. In the worst case, a compaction, two I/Os and another compaction all could happen while a client is waiting. A relatively simple solution to this would be a garbage collecting thread which performs deletes in the background. These compacted pages could then be scheduled for the disk in a low priority queue.

The ability to use large files ($\geq$ 2G) was not compiled into the Linux kernel we were using and this limited how large a disk we could seek over, even though it was unbuffered by the kernel. An easy-to-use solution to this would allow multiple partitions from the same disk. We felt this to be a cludge, however, and decided that large files would be commonly available soon enough.

Two items which were left unimplemented are checkpointing and the placement of groups as depicted in Figure 2. Checkpointing and recovery would be relatively simple due to the fact that any inconsistent data can be discarded. Adjusting each group's offset should not be difficult but, because turning the current one-way disk elevator algorithm into two-way was felt to be potentially complex and bug-prone, this was left unimplemented due to lack of time.

Two longer-term extensions are multiple disks and multiple pages sizes. Multiple disks would allow parallel access, especially to popular groups. Multiple page sizes would permit larger objects, like active cache tables, and potentially better hit rates based on the popularity of the group.

7 Conclusion

Dámelo achieves its goal of being a fast, easy-to-use object cache where its prime tuneables, grouping and memory usage, work effectively. Even under tests accessing objects randomly, it doubles the performance of ext2, a common example of a Fast File System. With better grouping, it bests ext2's average read latency by six times. It gains this speed primarily by keeping its meta-data in memory, which it can do because of its data's cache-only nature, and by placing related files adjacently on disk.

Hopefully, Dámelo will become the file system for the next popular web cache. Check it out at *www.damelo.org*!

8 Thanks

This thesis has been an extremely interesting, challenging, and rewarding project, and I would like to thank Remzi for all of his great ideas and good guidance!

References

[1] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms for log-structured file systems, 1995.

[2] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. The harvest information discovery and access system, 1994.

[3] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications, 1999.

[4] CacheFlow. Cacheos technology overview, 2000.

[5] David Dewitt and Jussi Myllymaki. Minirel database project, 1990.

[6] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Clusterbased scalable network services, 1997.

[7] G. Ganger and M. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files, 1997.

[8] Marshall Kirk. A fast file system for unix*.

[9] Q. Luo, R. Krishnamurthy, Y. Li, P. Cao, and J. Naughton. Active query caching for database web servers, 1999.

[10] A. Luotonen. Henrik frystyk nielsen, 1996.

[11] Carlos Maltzahn. Reducing the disk i/o of web proxy server caches.

[12] E. Markatos, M. Katevenis, D. Pnevmatikatos, and M. Flouris. Secondary storage management for web proxies, 1999.

[13] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log-structured file systems with adaptive methods, 1997.

[14] Matthew McCormick and Jonathan Ledlie. A fast file system for caching web objects, 2000.

[15] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, 1997.

[16] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO (USA), 1995.

[17] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15. Association for Computing Machinery SIGOPS, 1991.

[18] A. Rousskov and V. Soloviev. A performance study of the squid proxy on http, 1999.

[19] Alex Rousskov and Duane Wessels. Web polygraph, 1996.

[20] Thorsten von Eicken, Anindya Basu, Vineet Bush, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of 15th ACM Symposium on Operating Systems Principles*, pages 40–53. Association for Computing Machinery SIGOPS, 1995.

[21] D. Wessels. Squid internet object cache.

[22] D. Wessels. Squid internet object cache faq.

9 Appendix

9.1 Create Steps from Figure 3

a client makes request which proxy intercepts.

b proxy probes Dámelo groups for object's existence.

c Dámelo responds that it is not found.

d proxy forwards request to web server.

e web server responds with header and web object.

f proxy creates a buffer to copy web object into.

g proxy copies data from web server port to buffer and client's port. Note that this finishes the client's latency.

h proxy chooses a group (group 2) and calls Dámelo's create.

i group 2 picks a page with enough free space which is in memory. It picks logical page 5 which is in memory frame 2. It locks this page exclusively, adds the new web object, and unpins the exclusive lock.

j the internal slot array in the page notes the location of the new object.

k the group creates a new in memory file descriptor for the object. This thread is now finished with the create.

l sometime later when another thread needs a frame and frame 2 is at the head of the free list, it gets an exclusive lock on frame 2 and initiates writePage().

m after logical-to-physical frame translation, the request gets queued. After the disk has completed the request, it wakes up this second thread.

9.2 Code Fragment

```
#include <iostream>
#include "status.h"
#include "damelo.h"

int main () {

  Status status;
  int groupCount = 16;
  int memoryUsage = 128;

  Damelo *damelo = new Damelo ("/raw/raw/raw4", groupCount, memoryUsage, status);

  damelo->love (3);

  char fileName[20];
  sprintf (fileName, "somefile.gif");
  char dataIn[100];
  memset (&dataIn, 'a', 100);
  status = damelo->create (fileName, 3, dataIn, 100);

  int groupNumber;
  status = damelo->lookup (fileName, groupNumber);

  char *dataOut;
  int length;
  status = damelo->read (fileName, groupNumber, dataOut, length);

  status = damelo->release (fileName, groupNumber);

  damelo->hate (groupNumber);
  status = damelo->remove (fileName, groupNumber);

  delete damelo;
  return 0;
}
```

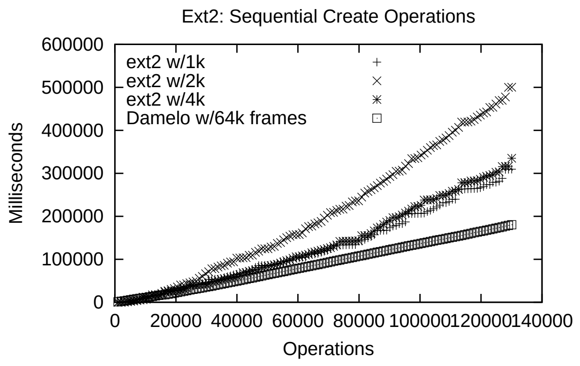


Figure 8: Ext2: Sequential Create

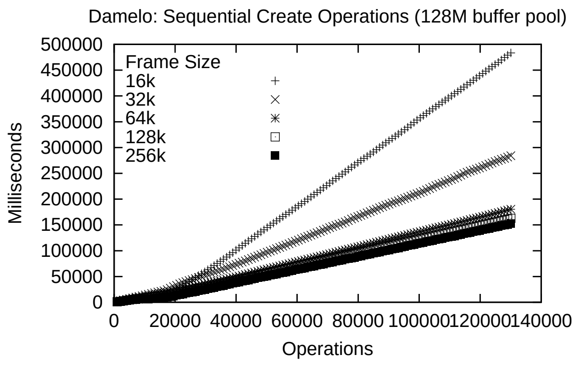


Figure 11: Dámelo: Sequential Create

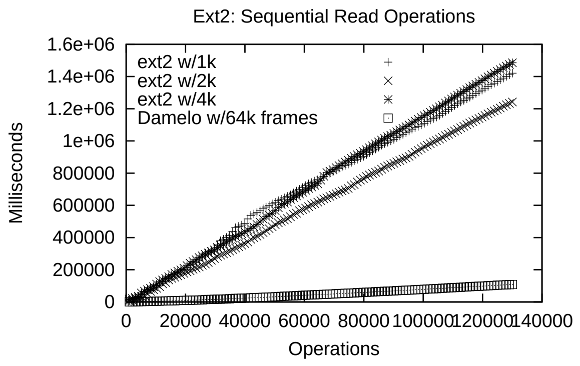


Figure 9: Ext2: Sequential Read

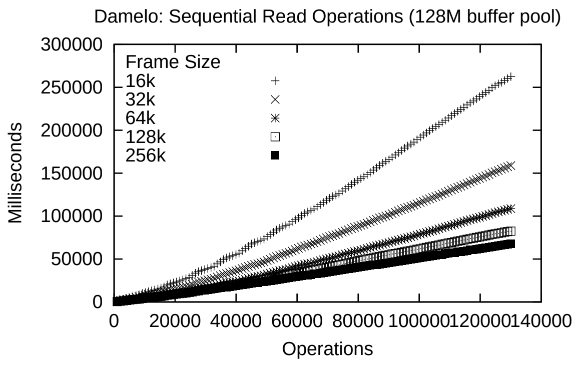


Figure 12: Dámelo: Sequential Read

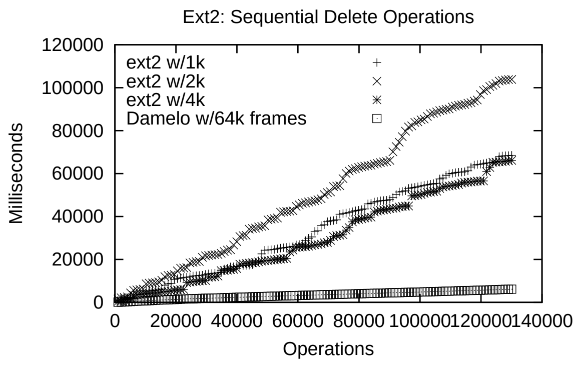


Figure 10: Ext2: Sequential Delete

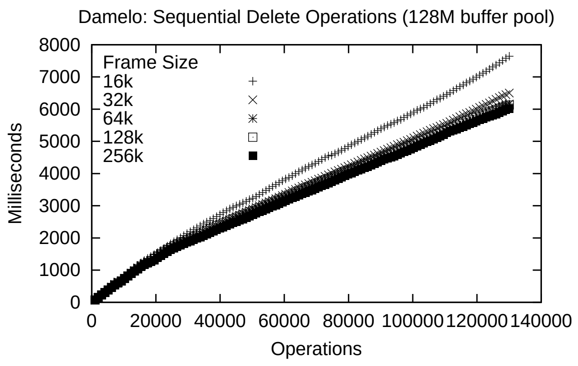


Figure 13: Dámelo: Sequential Delete

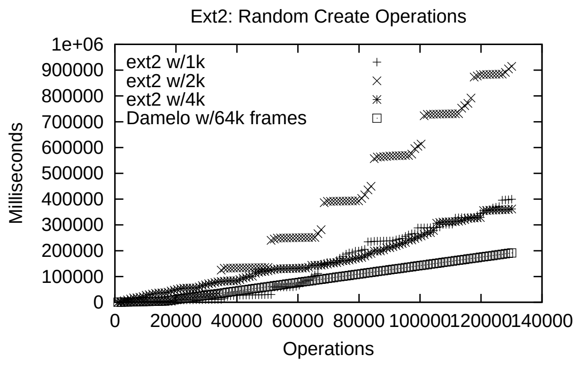


Figure 14: Ext2: Random Create

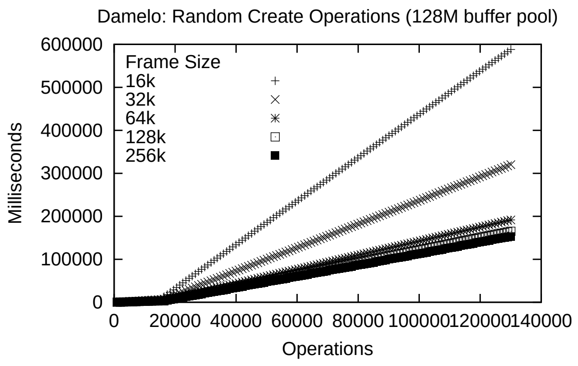


Figure 17: Dámelo: Random Create

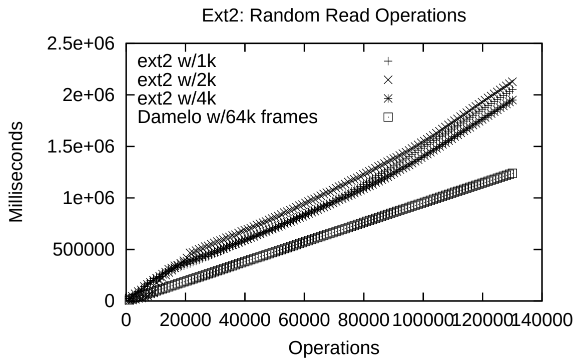


Figure 15: Ext2: Random Read

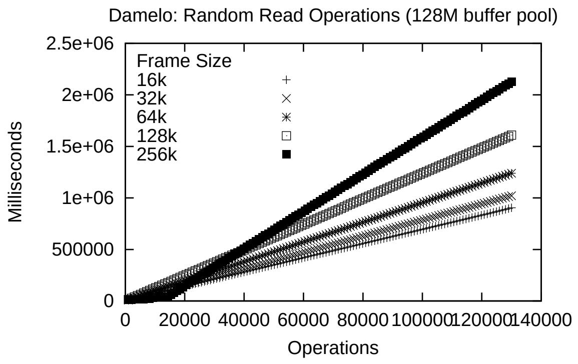


Figure 18: Dámelo: Random Read

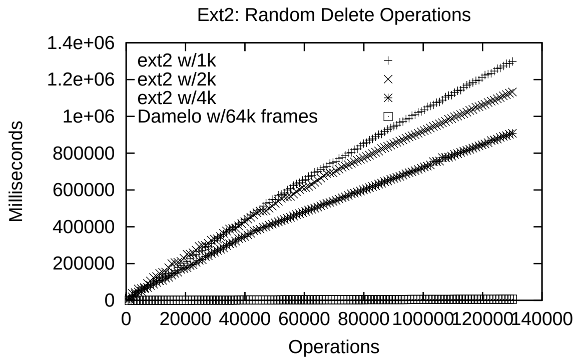


Figure 16: Ext2: Random Delete

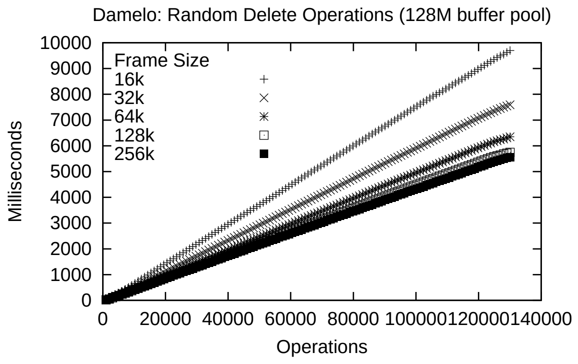


Figure 19: Dámelo: Random Delete

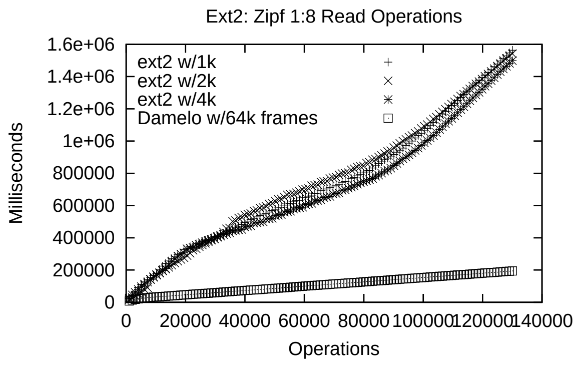


Figure 20: Ext2: Zipf 1:8 Read

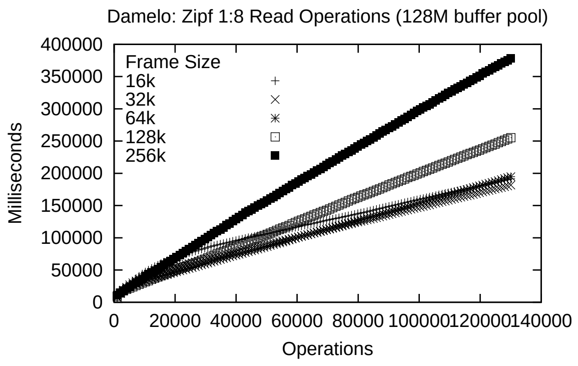


Figure 23: Dámelo: Zipf 1:8 Read

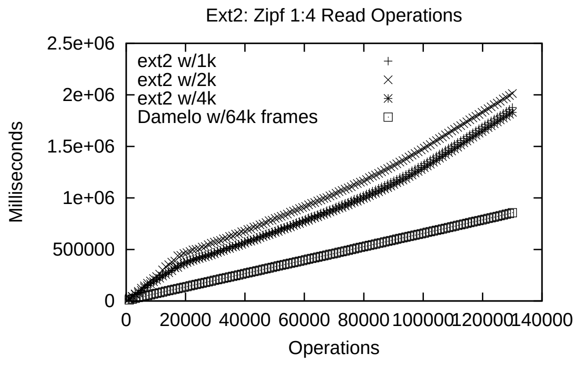


Figure 21: Ext2: Zipf 1:4 Read

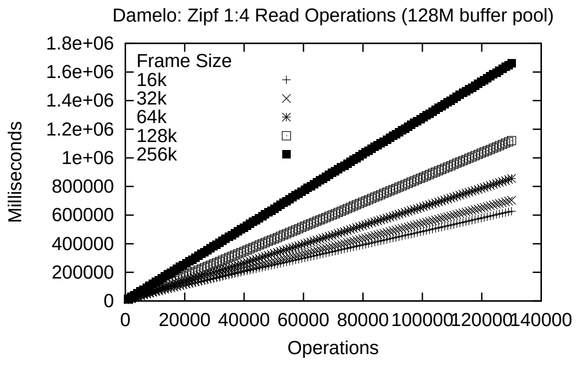


Figure 24: Dámelo: Zipf 1:4 Read

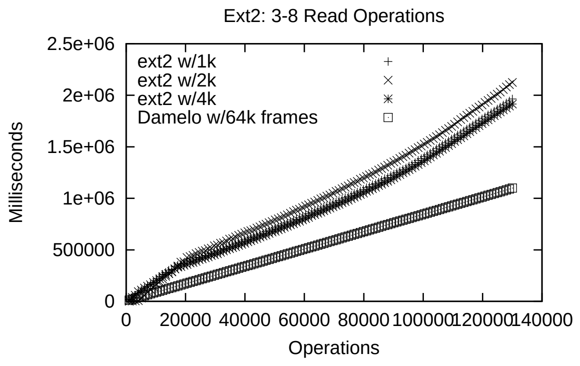


Figure 22: Ext2: Zipf 3:8 Read

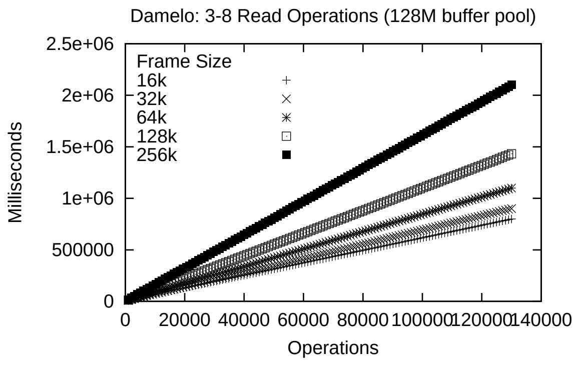


Figure 25: Dámelo: Zipf 3:8 Read