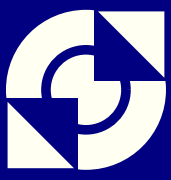
A Cache Odyssey

Peter Bosch

Pegasus paper 94–7
June 1994

University of Cambridge Computer Laboratory



University of Twente Faculty of Computer Science
Memoranda Informatica SPA–94–10

Pegasus, ESPRIT *BRA 6865, is a project of the University of Cambridge and the University of Twente.*
The project is aimed at the design of an operating systems architecture for scalable distributed multimedia systems and the development of a validating prototype, design and implementation of a distributed complex-object service and a global name service, mechanisms for the creation, communication, and rendering of fully digital multimedia documents in real time and in a distributed fashion (with support for full-screen digital motion video and digital hi-fi stereophonic sound), and the design and implementation of an application for the system — a digital TV director.

The Pegasus Papers can be obtained from the Pegasus Secretariat, Faculty of Computer Science, Vakgroep SPA, P.O. Box 217, 7500 AE Enschede, Netherlands.

They can also be obtained through the World-Wide Web:
`http://www.pegasus.esprit.ec.org/default.html`

University of Twente Memoranda Informatica

Pegasus Paper 94–7

A Cache Odyssey

Peter Bosch

June 1994

Abstract

This thesis describes the effect of write caching on overall file system performance. It will show through simulations that extensive write caching greatly reduces average file read latency. Extensive write caching reduces the number of disk writes and minimizes disk read/write contention. By taking a closer look at file system write semantics, it will also show that write optimized file systems are not the key issue for UNIX[1] like file systems. Write optimized file systems only reduce disk read/write contention without really solving the cause of disk contention. Simulations using the Sprite traces are used to guide the design of a client and server caching protocol for the Pegasus File Server (PFS). This protocol guarantees data persistency, without writing the data to disk, through replication.

[1] UNIX is a trademark of X/Open

1 Introduction

This thesis analyzes the UNIX workload in order to construct and evaluate caching algorithms that minimize file-system operational latencies. Based on the analysis, caches for the Pegasus File Server (PFS) [Bosch93] are designed. The analysis uses the Sprite traces [Baker91, Hartman93a, Hartman93b]. The traces are used to run simulations.

In the last decades many file systems have been designed and implemented. Each file system has its own characteristics and performs best for its design goal. There are several performance measures for a file system. File systems are designed to optimize storage layout, to provide sustained storage throughput, low latency read or write operations, minimize the amount of storage used, high reliability, high availability, or a combination of these. Each of the measures are optimized because the designers of the file system expected that most users working on that specific file system would require such an optimization. For example, if it is expected that many users will store large files that are rarely read, minimized storage layout and low write latencies are important. If it is expected that files will be heavily shared, the file system needs to be optimized for that case. A typical computer science workload is a so-called UNIX workload. This workload is characterized by a low rate of write sharing, high overwrite rates for new data, moderate persistency requirements, low program execution times and many files owned by single users [Burrows88, Ousterhout85, Baker91]. To optimize for such a workload, average read and write latencies must be low, otherwise the file system becomes the bottleneck in program execution. New data may be cached in memory before it is actually permanently stored on disk with the hope that the user will either delete the file and therefore make the need for the write operation go away, or that the user will overwrite the data with new data. In that case the data does not have to be written to disk. Since applications do not bother to check if data has safely arrived on permanent storage, low write latencies are possible.

To design an optimal caching strategy for UNIX workload, various caching experiments are tried. The results of these experiments are used to design PFS caches. First, in Section 1.1 a general file system introduction is given that will explain the basic functionality of a file system, followed in Section 1.2 by a description of all modules that are directly or indirectly associated with file systems. Next, Section 2 describes other work in the same area, and describes in some detail Pegasus and PFS. This section is followed by a description of the experiments and the results in Section 3. Based on the results, a caching algorithm is constructed that is explained in Section 4.

1.1 File Systems

A file system manages files. Each file can be separately identified and holds a sequence of data that persists even when the file system is not running.

Files hold information for users who invoke applications to read, modify and write files. A datum is a structured entity (record) or an unstructured entity (word of fixed size). The simplest model is a file system that manages files that are sequences of unsigned 8 bit words.

Users present a file-id, a representative of a file, to a file system and request the file system to perform an operation on a file. Since it is hard to distinguish between files

just on their file-id, file systems provide a way to associate a name to a file-id, a *file name*. Applications present the file name to a file system, which maps the name to the internal file-id. Usually, multiple file names can name the same file. If a file system grows, it can hold many files and when there is no way to order the files in some sort of hierarchy of related files, listing all the files can be cumbersome. Therefore, file systems order files in *directories*, a special file that contains either a reference to other directories or to files. An operation on a file is invoked by presenting a *path name* to the file system. A path name is a list of directory names followed by a file name and separated by a separation symbol. The file system traverses the path name, retrieves the file-id associated with the file name and executes the operation on the file. A name space is the closure of all path names and files. Name spaces are usually implemented in terms of files by the file system itself. To alter the name space, an application requests the file system to add, modify or delete a structured entity to a directory. Users are not allowed to perform the operation themselves because it is convenient if the name space closure is consistent for all times. If all files are accessible through the name space, the name space is consistent. Violation of this rule leads to un-referencable files.

In terms of operations, name spaces are implemented by the file system through read, modify and write operations on directories. User application requests on files are also in terms in read, modify, and write operations. So, the only important functions of a file system are read and write operations of (structured or unstructured) data to files.

1.2 File System Modules

A file system is built from some building blocks whose individual performance determine the overall performance. At the bottom layer there is a disk subsystem. Usually this disk subsystem consists of one or more high speed disks that can be organized in a RAID configuration [Patterson88]. A RAID system is a set of parallel disk with added redundancy. Data is written to all disks in parallel to increase disk throughput. The disks usually perform at a speed of 1–5 megabytes/s each and have an average data access time of 10–20 milliseconds. Some systems also provide a large but slow archival subsystem that runs at lower speeds and can have data access times up to several tens of seconds. If data is not accessed frequently, it is moved automatically to the archival subsystem, and the disk subsystem can act as a *cache* for the archival subsystem.

Data is moved from and to the disk subsystem through a memory cache subsystem. A cache is best described as a file system module that keeps a copy of data while the original is stored on a device that is "slower" than the cache. Caches usually do not provide persistency guarantees. A cache usually consists of some tens to hundreds of megabytes of ordinary RAM. The data access time of RAM is a couple of orders of magnitude higher than the access time of a disk, which means that if data can be served from file system cache, the read latencies are greatly decreased. Usually caches employ a simple LRU replacement policy to ensure that data that is frequently accessed is always available in memory. The file system throughput is thus increased greatly since most data does not have to be retrieved from the disk when applications request data. New data is usually written to the cache before it is written to disk. In UNIX [Ritchie74] clients do not have to wait for data to be written to disk; it is flushed asynchronously to disk. If client application want to make sure data has safely ar-

rived on disk, they can issue a FSYNC(2) system call. Cache replacement policies are built up from a statistical point of view.

Today, cache and disk subsystem are often put in a separate machine that is accessed over a network by clients. NFS is an example of such a network protocol [Sandberg85]. Instead of many independent file spaces on each machine, one unified file space is used. This file space is only implemented on the dedicated file system machines. Data sharing is simplified since those users that want to access the same file space simply use the same file systems. These dedicated machines can use all their available memory for caching and cache hit rates can be high. Further, only a small number of specialized machines need to be bought that can be equipped with large quantities of memory and disk.

Most client machines come with large amounts of memory and a disk. This memory is used to run client applications. So, in most cases binaries and data are loaded over the network to the client machine, the data is processed locally and the results are transmitted back to the remote file server. Most of the client memory is not used for file system caching, which means that if an application is run twice, data is read from the file server again and the network connecting both machines is used twice. Read and write latencies are thus increased by network transmission time. If there are many file system clients on the same network (e.g. Ethernet [Schoch82]), network contention can seriously influence file read and write latencies.

Network load (and server load) is reduced by making use of client caches. A *client cache* is a part of the file server that is integrated into the client's machine. It is as if part of the file server runs locally, increasing performance. Client caches are synchronized with the file server either through a cache consistency protocol [Popek85], by making the files immutable [Mullender89, Schroeder85] or by ignoring consistency problems if most files are not shared. The latter approach seems to work reasonably well in UNIX environments as is demonstrated by the number of sites that run NFS today.

2 Related Work

This section lists issues related to the design of low latency UNIX file systems. Section 2.1 presents an overview and describes UNIX workload in some detail. Next, since the goal to design a low latency caching strategy for PFS, an overview of the Pegasus project is given in Section 2.2 and the Pegasus File Server (PFS) is explained in Section 2.3.

2.1 Unix File System Issues

Since program execution times are low in UNIX systems, low file read and write latencies are important. Read latencies are kept low by caching frequently accessed data in RAM. Ousterhout [Ousterhout85] and Baker [Baker91] showed that extensive caching greatly improves file system read performance. Write operations have a different property. In general, when an application writes data it is only interested in persistent data store. Current UNIX systems only guarantee data persistency when the data is a couple of seconds old (unless the applications calls FSYNC(2), which immediately updates the disk). It buffers the newly created data in memory for some time (usually 25–30 seconds) before it writes it to disk or to a remote server. This has the advantage that if data is removed within that period, it does not have to be

written to disk. Most UNIX applications can live with this situation. Since the buffer period is usually small and system crashes are rare, applications count on not losing the data they write.

As processor speeds increase by orders of magnitude, more file system throughput is required. The file system is becoming a bottleneck in the program execution since disks are not getting faster with the same pace. UNIX semantics requires data to be written to disk after the small period, which can introduce a burst of write traffic, causing long queuing delays. UNIX FFS [McKusick84] is a read optimized file system. This means that if the file system is updated, the locations where the file system performs the update are chosen for good read performance when the file is read back from the file system. Since the focus is on read performance, write performance is less of interest, and therefore may not be as good. This can cause long write queues, which in turn delay read operations. By using large read caches to reduce read latencies [Ousterhout85] the disk executes more write operations than read operations. This has prompted some researchers to develop write-optimized file systems. Sprite LFS [Rosenblum91] employs large sequential disk writes to a *file system log* to improve disk write efficiency. In a log-structured file system, updates are appended to the log rather than changing the file system blocks in place. Indexing tables are maintained to retrieve file system meta data from the log since that information does not have a fixed location anymore. However, log-structured file systems suffer from two problems. UNIX FFS has shown that file read performance is improved if logically related files are clustered in the same areas on disk. Sprite LFS cannot guarantee this and read latencies are influenced by extra disk seeks. Sprite LFS masked some of this by using large read caches. However, in Sprite LFS there is still the possibility that a large write operation to disk is in progress while a read request can not be satisfied from the cache, causing the read to wait for the write to complete. Since write operations on Sprite LFS are usually large, file read latency is increased for those files that are not in the cache and need to compete for the disk with writes.

As stated earlier, the focus of this thesis is to design an optimized caching strategy for UNIX workload. This workload has several characteristics pertinent to file system design [Baker91, Ousterhout85, Burrows88]. In UNIX file systems in research environments, most newly created files (70%) are deleted within 30 seconds and most newly created bytes (80%) are deleted within 6 minutes. Many files are created and are immediately thrown away by applications such as editors and compilers. This also means that long files live longer than small files. By delaying writes for 6 minutes, much less data needs to be written to disk.

Roughly 70% of all client-to-server write traffic is generated due to the 30 second sync timeout in the client file system. Further, in the same article it is noted that client caches of 200 kilobytes to 23 megabytes, reduce server read traffic by 40%. Using client read caches also decreases file read latency: the data is already in local memory and the server does not have to be contacted.

Files are generally not (write) shared. Burrows [1988] reported that only 0.53%– and Baker [1991] reported that 0.34%– of all file server requests result in conflicts (shared read/write). Hence, if file systems are extended with client caches, fewer demands are placed on the file server and more clients can be served by the same server.

When client caches are used, multiple copies of the same data are available in the

system. If a file is read/write shared, inconsistencies may occur. Since files are rarely shared in UNIX, caches can be synchronized by a cheap cache consistency protocol performance is optimized for non-shared files. A file service (including client caches) that offers file semantics equivalent to a *regular register* [Lamport85] with respect to individual reads and writes would be good enough. In regular registers, if a read overlaps with a write, the read returns either the old or the new value. Such semantics are equivalent to the semantics offered by a single-site file system. If an application needs a higher degree of file consistency (e.g. an *atomic register*), it must use a locking protocol. Echo [Mann93] and MFS have shown that a token-based mechanism is a simple and efficient way to implement such a regular register.

On average files are written 1.6 times in a segmented log-structured file system due to log cleaning operations, which copy and compact the data to free segments. Since files are forced to disk after 30 seconds even though most files and data will be removed within several minutes, segments at the end of the file system log are cleaned often [Rosenblum91]. The cost of log-cleaning is proportional to the number of unstable bytes (bytes that are deleted or overwritten). Minimizing this number reduces the amount of log-clean operations. This can be accomplished by using large write cache timeouts.

Log-structured file systems have the advantage that the file system that resides on disk is always consistent. It is never the case that, while updating a part of the file system, several data structures on disk are inconsistent, as is the case in UNIX FFS. If a log-structured file system fails, its state can easily and quickly be reconstructed from the disk file system. Large UNIX FFS file systems suffer from extensive recovery times.

2.2 Pegasus

The Pegasus project[2] investigates the integration of multimedia as a first-class citizen in the operating system. It aims at building a fundamentally new system architecture that treats all types of existing data (text, graphics, audio, and video) as first-class citizens of the operating system. Current computing systems treat multimedia as an *add-on*, which leads to systems that do support multimedia data but cannot really *handle* multimedia data. On a PC, for example, a multimedia application takes over the complete machine, and no other job can run. When multimedia applications are started on conventional time-sharing machines such as UNIX, other jobs can be executed, but real-time guarantees cannot be given to the multimedia application, which leads to poor performance.

The Pegasus approach is a fundamentally new approach. All parts of the system know how to deal with multimedia data and know how to multiplex resources over the data streams. The Pegasus distributed system is built on top of a high speed ATM network. This network carries all information that is managed in the Pegasus system. Multimedia devices such as cameras, displays, and audio equipment are directly connected to the network. Computing devices themselves are decentralized and employ an ATM *Desk Area Network (DAN)*. Memory, processor and I/O devices communicate through this small ATM network.

Multimedia data is delivered in this system by setting up a virtual circuit between source and sink. This means that ATM cameras send their data directly to the ATM

[2]Esprit BRA 6865, September 1992–1995, University of Cambridge, UK, and Universiteit Twente, NL.

display without first having to send the data through some computing device. The major advantage is that bulk data is sent over system busses only once. If multimedia data needs to be processed before it is rendered on a display, the data is sent to the processor by setting up a virtual circuit directly to the processor. The processor sets up a second ATM virtual circuit to deliver the data for further processing, storage or display.

The Pegasus system itself consists of 4 major components:

- ATM (multimedia) devices. Hardware devices that are directly plugged into the network.
- A new micro-kernel called Nemesis. This kernel gives a *Quality-of-Service* to threads. With this QoS, the system is able to provide soft real-time guarantees.
- A naming system to bind all system objects in a naming hierarchy.
- The Pegasus File Server, which is able to store and retrieve multimedia data in a timely manner and is able to support ordinary UNIX file I/O. PFS is explained in more detail in Section 2.3.

More about the Pegasus system can be found in [Mullender94, Hyden94, Barham94] and [Pratt93].

2.3 Pegasus File Server

In the basic PFS architecture, the file server is divided into client agents, server machines, and a disk subsystem. Figure 1 shows this architecture. A more detailed description of the Pegasus File Server can be found in [Bosch93].

The Pegasus File Server stores and retrieves both continuous media and ordinary UNIX files. By using large blocks for continuous media, the file system makes efficient use of the available disk bandwidth (for example less time is spent seeking). Other large blocks are used as segments in a segmented log-structured file system much like Sprite LFS or BSD LFS [Seltzer92]. The result is that PFS provides a write optimized file system for UNIX like traffic.

- The disk subsystem is responsible for reading and writing blocks from and to disks. The disks are used to record file system updates and to cache archived data. Disks are attached in a parallel RAID-5 configuration which (theoretically) increases the available disk throughput by a factor 4. In a later stage, the disk subsystem will also be made responsible for an archival storage system much like in [Pike90].
- A high-speed file server. The files are organized in a segmented log-structured fashion for high write performance. The file server prepares segments that fit exactly across disk stripes and are written to the $4 + 1$ disks in one write operation. Since data is written as complete stripes, a separate read to recalculate the XOR disk is not needed. Continuous media data is stored and retrieved out-of-band, and does not *flow* through the actual file system (it does not make use of the caching facilities). Initially there will be no disk bandwidth reservation scheme. Instead, PFS relies on the availability of enough bandwidth. If this

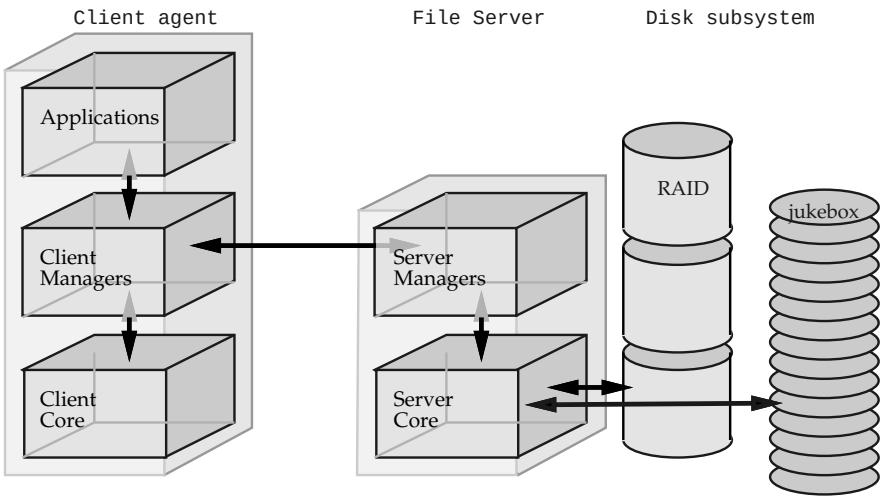


Figure 1. PFS architecture

	PFS	Sprite LFS	FFS
Caches	Large write	Large write	Small write
Optimized for	Continuous media, write	Write	Read
Disk layout	Log	Log	Cyl. groups

Table 1. PFS, LFS and FFS differences

strategy does not work, a bandwidth reservation algorithm will be introduced later. Lastly, a large portion of the server memory is used for read and write caching ordinary (i.e. small) files. The file system runs an aggressive write cache policy: it tries to filter out as many write requests as possible before the files are written to disk by exploiting the overwrite nature of UNIX workload. The effect of write caching is the topic of this thesis and is described in Section 3; the design of the write caching strategy is explained in Section 4. The file server is powered by a UPS to protect volatile data from being lost during a global power failure.

- Client agents provide the client's interface to the file system. Client read and write caching is integrated in the client agent. Client cache consistency is maintained through a token-based mechanism much like the Echo file system or MFS. Tokens have a lifetime to guarantee progress even if client machines fail or the network partitions. If a token is not yet expired, client agents may serve client requests from their cache. The file server can call back to the client agent to tell the client to discard a file from its cache. The client agent runs an aggressive write cache policy much like the server write policy. Client write caching and token management is explained in Section 4.

Table 1 lists the basic differences between PFS, Sprite, and UNIX FFS. UNIX FFS is the only file system that has optimized read operations, both PFS and LFS optimize write operations. PFS is also able to deal with continuous media data.

Pegasus File Server status

The Pegasus file server currently runs on a DecStation 5000/240 and is connected to the outside world through an ATM network. Currently, only a single disk is used. File server cleaning is not yet implemented. Server caching is implemented, client caching is not yet implemented and is part of this thesis work. The servers are integrated in UNIX environments through a client daemon on the UNIX machines that communicates with UNIX clients through the vnode interface [Kleiman86]. No effort has been made to migrate the client cache into the UNIX kernel due to lack of manpower. Such a migration would improve efficiency.

3 Simulating Extensive Write Caching

File systems perform two basic operations: read and write. Since program execution times in UNIX are low, read and write operation latencies must be low. The obvious bottleneck in a file system are the disks. The data access times of disks are orders of magnitude higher than the access time of state of the art networks or RAM. So, to optimize file system operations, the disk must not be in the data path. Read latencies are minimized by making use of extensive read caches. Write latencies are minimized by updating the disk asynchronously. The Sprite traces showed that many new bytes are deleted from the file system while the bytes are still young. UNIX file systems employ a *write cache timeout* of 30–60 seconds timeout to guarantee persistency of data after a small period. The main problem with the small write cache timeout is that most of the written data will soon be deleted, which involves more disk operations. If a read operation can not be satisfied in the file system cache, the data needs to be read from disk. But this disk is used extensively for write and overwrite operations: as a result, read latencies are increased. I have called this *disk read/write contention*. There are two approaches to reduce this contention. Sprite LFS and BSD LFS have optimized file system write performance, which reduces the length of the disk queues. The other approach is delaying disk writes somewhat longer to reduce the amount of data that needs to be written to disk. Careful caching using replication as is shown in Section 4 guarantees persistency of data even if the data is not written to disk.

This section analyzes the effects of this extensive *write caching* on file system performance.

If data is buffered for a longer period in memory before it is written to disk, less data will be written to disk. In the ultimate case, the disk only services read requests that do not hit in the cache, and read latencies are minimal. However, if written data is buffered in the cache, more read operations miss in the cache since the *write cache* occupies memory blocks that also could have been used for read caching[3]. The experiments that are described in this section try to find an optimum for the *write cache timeout*. This parameter defines how long a cache block may be dirty.

The Sprite project have measured their file system and have made the traces publically available[4]. Since the Sprite workload is similar to the workload that is expected for the non-continous media part of PFS, the Sprite traces are used to simulate extensive write caching. Results previously gathered with the Sprite traces are used

[3] In the extreme case, a disk is only there to make a whirring sound and provide an extra blinking light – RG.

[4] ftp://sprite.berkeley.edu/sosp_traces

in the analysis without further validation. The traces consist of 8 days of general workload. The workload consists of all user operations on the Sprite system that involved file system activity. On average there were 30 different users working on the system from 40 workstation, all running the Sprite operating system [Ousterhout88]. The main server contained 128 megabytes of main memory. In the simulations that follow, only file system activity from the main server is measured. All user open, close, lseek, truncate, delete, name space, and attribute operations which are serviced by the file server are recorded. The amount of data read and written can be calculated through lseek and close operations. A detailed description of the daily workload is described in Baker91 [Baker91], and Hartman93b [Hartman93b]. Hartman93a [Hartman93a] describes in detail the structure of the trace files themselves and the simulation suite that comes with the traces.

What will be shown in Section 3.1 is that write caching can reduce the amount of data that needs to be written to disk enormously. However, it will also be shown that writing less data is not the key issue in building a low latency file system. Instead, it will be shown that write caching reduces disk contention that is introduced if clients and server only buffer for 30–60 seconds. It is argued why log-structured file systems are not really the answer to the growing I/O bottleneck. If new data is kept in memory longer, disk contention reduces and read latencies are minimized.

Based on these observations, in Section 3.2 write strategies are examined that give precedence to read operations to reduce read latencies even more. Disk write operations are preempted to give precedence to read requests. It will be shown that such a strategy can reduce file read latency substantially at the costs of severely delaying writes.

3.1 Write Caching

In a UNIX client system, data is buffered for up to 30 seconds before it is written to the server or disk. Although many bytes are overwritten in the first 30 seconds of a file's lifetime (30–63%), many more are overwritten before the file is 1,000 seconds old (60–95%) [Baker91]. In UNIX file systems, it means that most updates will be written to disk even if it is likely that the newly written bytes will be deleted. A part of the available bandwidth is wasted since most of the data that has just been written is immediately removed.

In the first simulation, the maximum possible reduction of writes and the size of the write cache is measured, using a write cache without memory bounds. The time a dirty block may reside in the write cache is variable. By steadily increasing the write cache timeout, it takes longer for the newly created data to reach the disk but it is also more likely that data is overwritten before it reaches the disk. The simulator replayed part of the Sprite traces and recorded the effect of write buffering in the simulated write cache. In this simulation it is not assumed that files are written sequentially as in the Sprite and Unix 4.2 BSD traces [Ousterhout85]. The BSD and Sprite traces assumed files are written sequentially and calculated the overwrite factor by examining the time of last modification in the file's inode. Every second the contents of the write buffer is examined, and when the data is flushed to disk, the operation is recorded.

Figure 2 shows the average overwrite factors as function of the write buffer time. First, it shows that many bytes are discarded in the first second. This is probably caused by applications that create a large file and immediately truncate the file again.

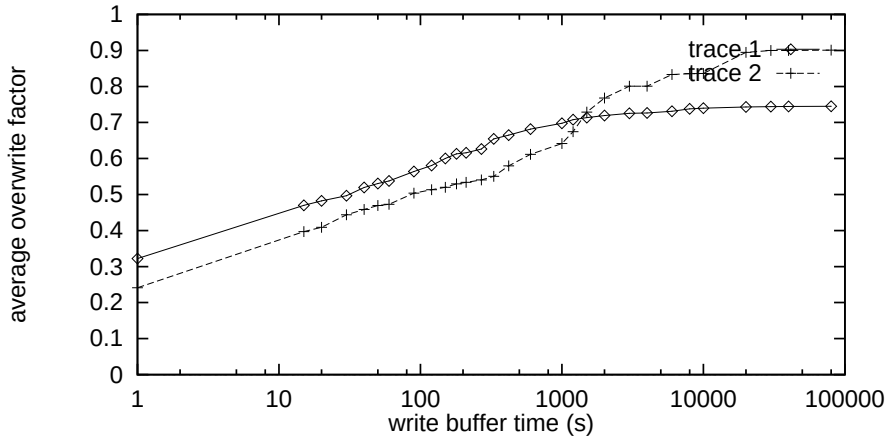


Figure 2. Overwrite factor as function of write buffer timeout

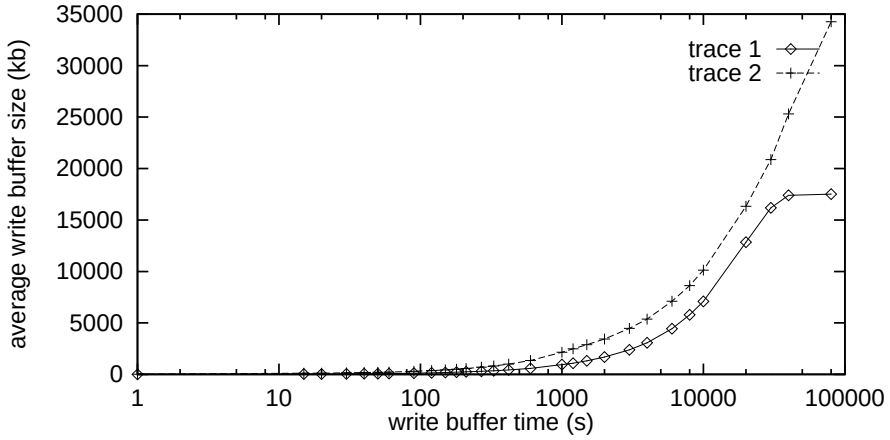


Figure 3. Write buffer size as function of write cache timeout value

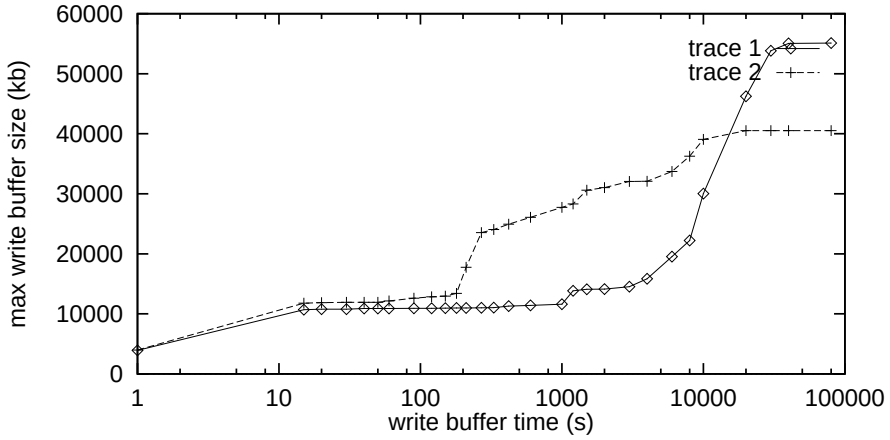


Figure 4. Maximum cache size

It shows that there is a steady growth in the overwrite factor to about 90% until the buffer time out is ≈ 1,000 seconds. So, if data is buffered for 1,000 seconds or more, the amount of data that needs to be written to disk is only 10–20% of the data originally written. Buffering more than 1,000 seconds has a substantial memory cost as is shown in Figure 3. The average write buffer size is not really large (100 to 500 kilobytes for 100 seconds write buffer timeout, 1 to 5 megabytes for 1,000 seconds write buffer timeout) but since traffic is bursty, maximum write buffer sizes are also important as is shown in Figure 4. During peak times, write cache size grows to 50 megabytes or more when the write cache timeout equals 1,000 seconds. Given today's file server memory sizes (256 megabytes or more), reserving a maximum of one fifth of this memory for write caching purposes is not unreasonable. Sprite reported that a block is unreferenced on average for 48 minutes before it is replaced in their 128 megabytes cache. Since most caches employ a simple LRU strategy, using a write buffer timeout of 1,000 seconds instead of 30–60 seconds does not influence the read cache hit rates: the data would also have been in the cache if a smaller write buffer timeout is used. Since the overwrite factors are similar to the results in Baker91 [Baker91] only 2 traces are examined. It is assumed that all other traces behave as those reported.

So, if the file server buffers data for some time to discard as many bytes as possible in the cache, less data is written to disk and more disk bandwidth is saved. The analysis shows that it is possible to keep written data in memory much longer (both on clients and servers). But, since the amount of data that is actually written to disk per day (Sprite reported on average ≈ 7 kilobytes/s) it seems that there is little any need for extensive write caching to reduce the amount of disk writes.

A closer look at maximum write cache (Figure 4) sizes shows why log-structured file systems are important. If data is only buffered for 30–60 seconds before it is flushed to disk, maximum write buffer sizes of 10 megabytes exist. When this data is flushed to disk in a UNIX FFS, the disk is busy for a long period, given that the file system can write at a few hundred kilobytes/s. Low write performance increases average read latency since reads are stalled by the (slow) writes. A log-structured file system improves the write performance. Combining updates into single large writes reduce the busy times of a disk for write operations but does increase the possibility of increased temporary file read latencies and an inefficient storage layout for read operations. On the other hand, since disk contention is reduced, average read latencies probably decrease.

A simulation was set up that actually measured the read latency in a simulated write optimized file system that uses the same characteristics as PFS. When a file is updated, it is written as a whole (that is, the part that is in the cache) to disk. This strategy approximates whole file storage since Sprite reported that most files are written as a whole. File read performance is maximal since the files are stored consecutively. The system uses a disk similar to a Digital DSP3105 disk [Digital92] which has a transfer speed of 2.6 megabytes/s, an average data access time of 15.1 milliseconds and is equipped with a variably sized cache. The Sprite traces only record at most 2 consecutive days, so only cold caches are simulated. However, simulating two consecutive days still resulted in cache hit rates up to 70–90%. What is measured is the total delay when reading blocks from disk (including queue time). It is assumed that read requests that are satisfied from the cache take zero time to complete. If a read operation misses in the cache, the data is read from the disk. If the disk is already busy servicing another request, the current request is queued.

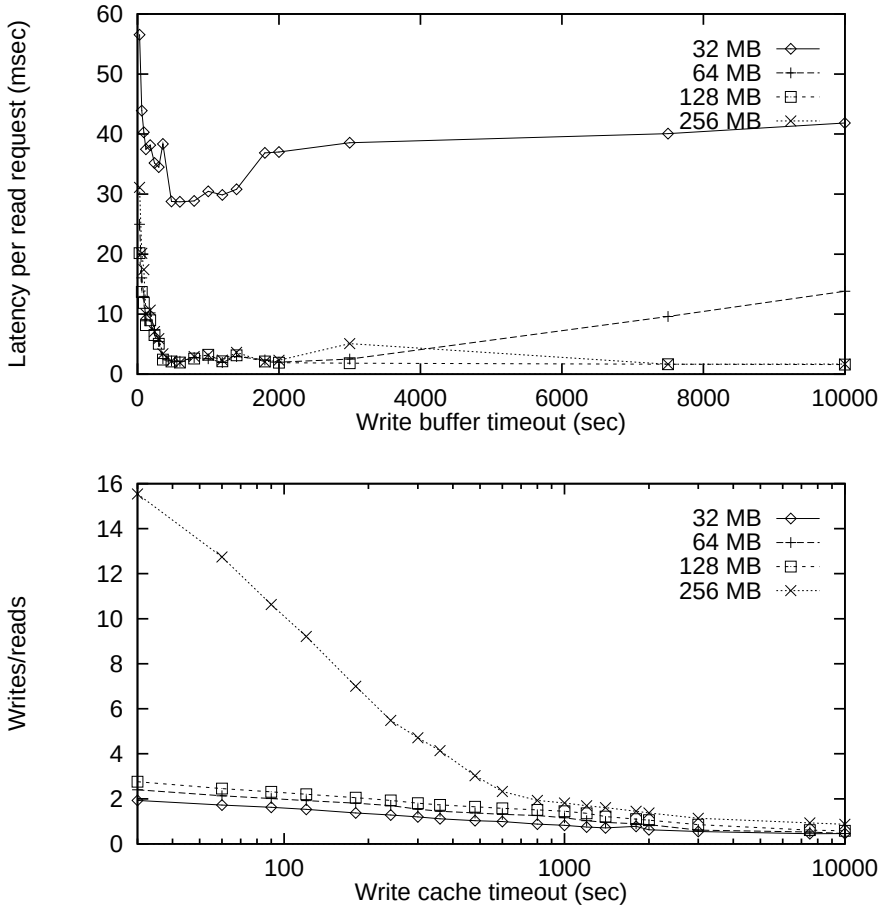


Figure 5. ***Trace 1***

There are several parameters influencing the performance:

- Several cache sizes are simulated: 32, 64, 128 and 256 megabytes.
- Only 25% of the available memory may be filled with dirty data. Since there are more reads than writes, it is reasonable to reserve a maximum part for write caching. 25% seems an appropriate percentage. If there would be no hard limit, a single application can clear and lock the complete cache. The next cache miss (in fact, the next file system request) would suffer badly. First, the dirty data needs to be written to disk. Next, the non-dirty data needs to be removed from the cache. Finally, the requested data can be read from disk. Therefore, flushing the dirty on before-hand to disk may result in lower read latencies.

In the next subsections, the effects of extensive write caching is explained in detail for all traces.

Trace 1

Figure 5 shows the average read delay to read a 1 kilobyte block from disk. If there is only 32 megabytes cache space of which only 25% may be used for write caching, new data cannot be buffered longer than 1400 seconds. If data is buffered longer, the write cache will be larger than 8 megabytes, which will have the effect of a smaller write

cache timeout since data is forced to disk even if the new data is not old enough. The graph also shows that a cache of 32 megabytes is simply too small for this workload. Read latencies are between 90–30 milliseconds if new data is buffered less than 360 seconds. Larger cache sizes lead to much lower read latencies. Cache sizes of 64, 128 and 256 megabytes lead to exactly the same read latencies if the buffer time is less than 2,000 seconds. If write cache timeouts are larger than 2,000 seconds, the write buffer becomes larger than 16 megabytes and new data is flushed to disk even if it is not old enough in the 64 megabytes case. The extra read delay at $t = 3,000(sec)$ when a 256 megabytes cache is simulated, is caused by the whole file storage policy. All blocks associated with a file (dirty or non-dirty) are flushed to disk when either the timer expires or when the cache is full. This trace contains applications that updates non-dirty files after a long period and in the 128 megabytes case the non-dirty blocks are already discarded from the cache. In the 256 megabytes case, the non-dirty blocks are still in the cache and are flushed multiple times to disk.

Figure 5 also shows the read/write ratio as seen by the disk. First, it clearly shows that buffering for 30–60 seconds in not a good idea in this trace. There are many more writes than reads, especially for the larger caches. This leads to disk contention and increased read latencies. The read/write ratio decreases exponentially to one, showing that file systems are not write dominant if new data is buffered for longer periods.

This trace performs best by using a reasonably large cache and a write buffer time, larger than 360 seconds.

Trace 2

Figure 6 shows the average read delay in trace 2. It clearly shows that read delays suffer if the cache size is smaller than 128 megabytes. As is shown in the same figure, this trace is almost read dominant if there is a large cache and write cache timeouts are reasonably large. Extending the write cache timeout does not help in the 32 megabytes and 64 megabytes case since those simulations are dominated by the flushes due to the 25% write cache rule.

Trace 3

As is shown in Figure 7, trace 3 is also disk read dominant. If new data is buffered longer than 2,000 seconds, there are more reads than writes. If new data is buffered less than 2,000 seconds, the read/write ratio is not that high (when compared with trace 1). It means that average read latencies are not influenced by disk read/write contention. Only a larger cache decreases the average read latencies, indicating that there is only disk read/read contention.

Trace 4

This trace shows several effects. First, as is shown in Figure 8 large caches reduce the read latency and, makes the system read dominant if new data is buffered for more than 2,000 seconds. The graphs clearly show that a 32 megabytes cache is too small. The write cache suffers from the 25% rule which causes a fair amount of disk read/write contention. The 64 megabytes case suffers less. Read latencies are low if the write cache timeout is less than 1,800 seconds. At 1,800 seconds, the 25% rule comes into effect. In this simulation, the same effect is shown as the effect described

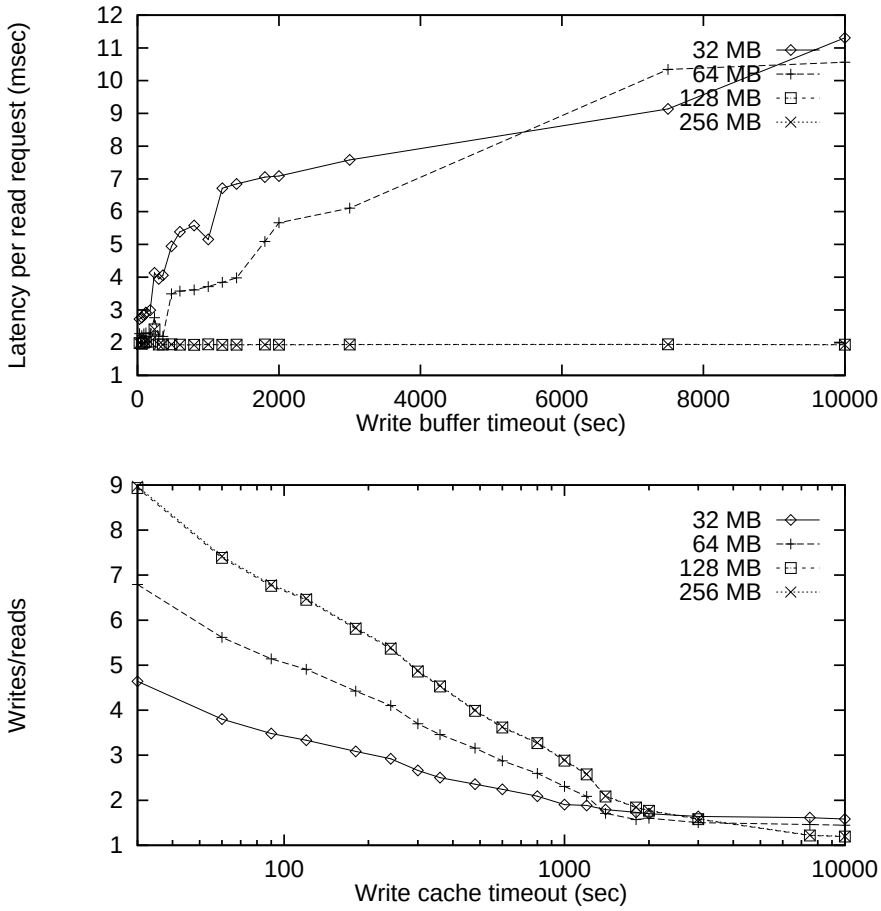


Figure 6. ***Trace 2***

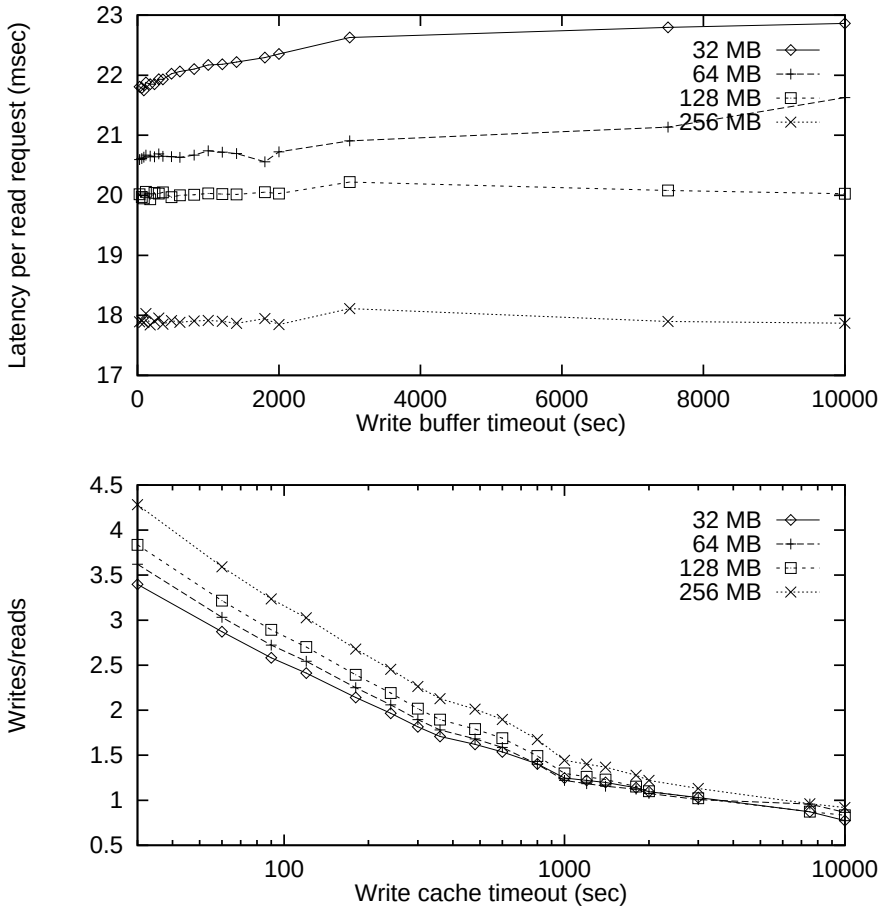


Figure 7. ***Trace 3***

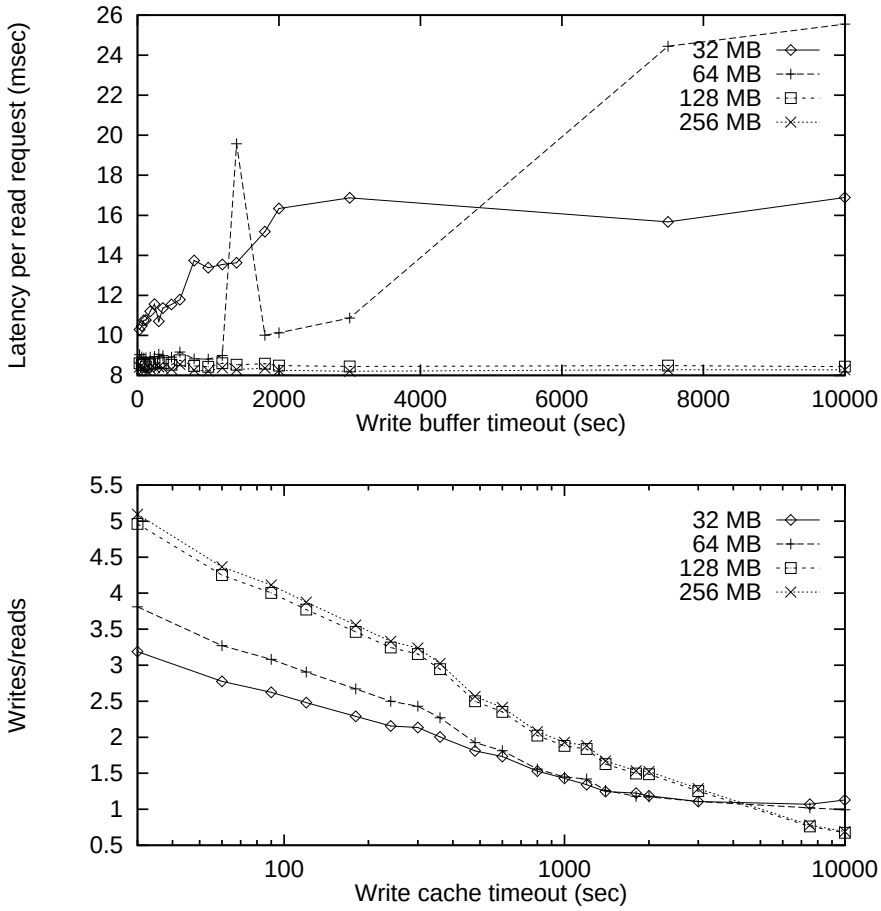


Figure 8. ***Trace 4***

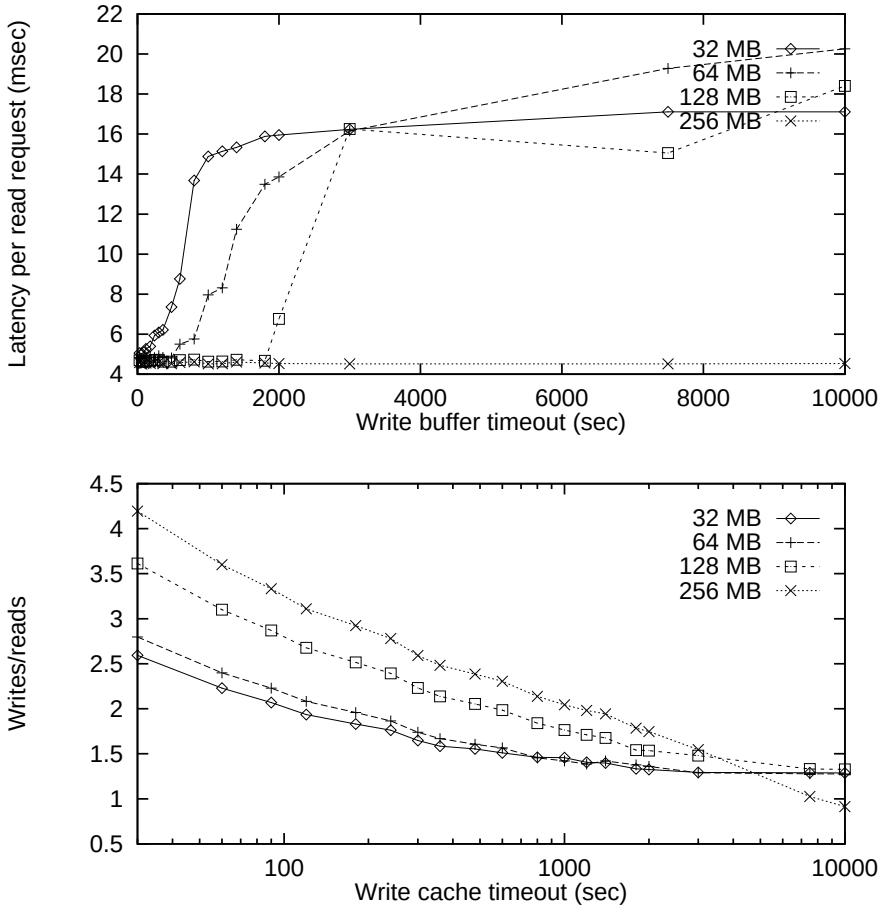


Figure 9. ***Trace 5***

in trace 1. Many non-dirty blocks are written to disk multiple times. Longer write cache times, cause the non-dirty blocks to be removed from the cache sooner and leads to less data written to disk.

Larger caches (128 megabytes and 256 megabytes) show that if there is enough cache available, the system becomes read dominant and there is hardly and disk read/write contention. Not many bytes are written in these simulations.

Trace 5

The last trace shows that large caches are the important factor for low latency file systems. In Figure 9, it is shown that all read latencies drop to $\approx$ 18 milliseconds, if the cache is too small. Too small caches lead to a write dominant workload. If the cache is 256 megabytes large, the system becomes read dominant if data is buffered for 8,000 seconds. Since the 25% rule does not come into effect in the 256 megabyte case, and there is hardly any read/write contention, read latencies are low for this trace.

Summary of the traces

There are several conclusions that can be drawn from the simulations. First, large caches are the key issue in building low latency file systems. If large caches cannot

be used, average read latencies are dominated by the disk's data access time. Second, extensive write caching reduce disk read/write contention since many bytes are overwritten in the cache. Third, while Sprite reported that large caches lead to write dominant file systems (as seen from the disk), the traces show that this is only true for write buffer time of 30–60 seconds. If new data is buffered for longer periods, the system becomes less write dominant (sometimes even read dominant). Therefore, a low latency file system needs to be read optimized.

3.2 Disk Queue Reordering

In the previous subsection it has been shown that extensive write caching can reduce read latencies. This subsection analyzes a model that tries to reduce read latencies even more. In the following simulations reads take precedence over writes.

To reduce read latencies, reads should never be stalled by write operations. The disk reorders disk I/O requests such that reads take precedence over writes and reads preempt disk write operations. Read latencies are reduced at the cost of delayed writes.

Ideally, a disk would maintain a separate read and write queue. The write request queue is only serviced when the read request queue is empty. Few (if any) disks can handle such a strategy[5], which implies that the file server must do the ordering of requests by itself. Large write requests are split into many small requests that are sent to the disk controller just before the previous small write request finishes. By carefully timing the operations, the assumption is that many small writes have roughly the same throughput as a single large write, since there are no seeks[6]. The advantage of this strategy is that a large write can be preempted at any instant to give precedence to a read request. This simulated disk is inserted into the simulation described in Section 3.1. In the following simulation it is assumed that an interrupt costs 30 milliseconds (15 milliseconds data access time and 30 kilobytes of new data). What is measured is the average read delay and the extra write delay caused by the interrupts. The simulations approximate a file server. If data is sent to disk, the simulation assumes that the cache is instantly non-dirty. So, if new data is buffered for longer periods at the disk, write buffering may reduce read cache hit rates that do not appear in the simulations. If the extra write latency is small, these effects can be neglected.

Trace 1

Minimized read latencies are up to 26 milliseconds when the cache is too small. There is a huge read/read contention on the disk since, by definition, reads are not delayed by writes. It clearly shows that 32 megabytes cache space is not enough. The maximum number of interrupted writes for all traces is 6,000. This number reduces exponentially to ≈ 500. The influence of interrupts on average read latencies is minimal ($\ll 1$ millisecond). Figure 10 also shows that read latencies are low by using larger caches. Write operations are delayed somewhat, as can be seen in the bottom graph of Figure 10. Only if data is buffered for more than 2,000 seconds, write delays are

[5]The DEC DSP3105 [Digital92] has a mode to reprogram the drive. Read requests can be given precedence in the drive.

[6]Some disks are able to start writing in the middle of a track if a complete track is written. Dividing a large write operation in many small write operations may eliminate this disk optimization.

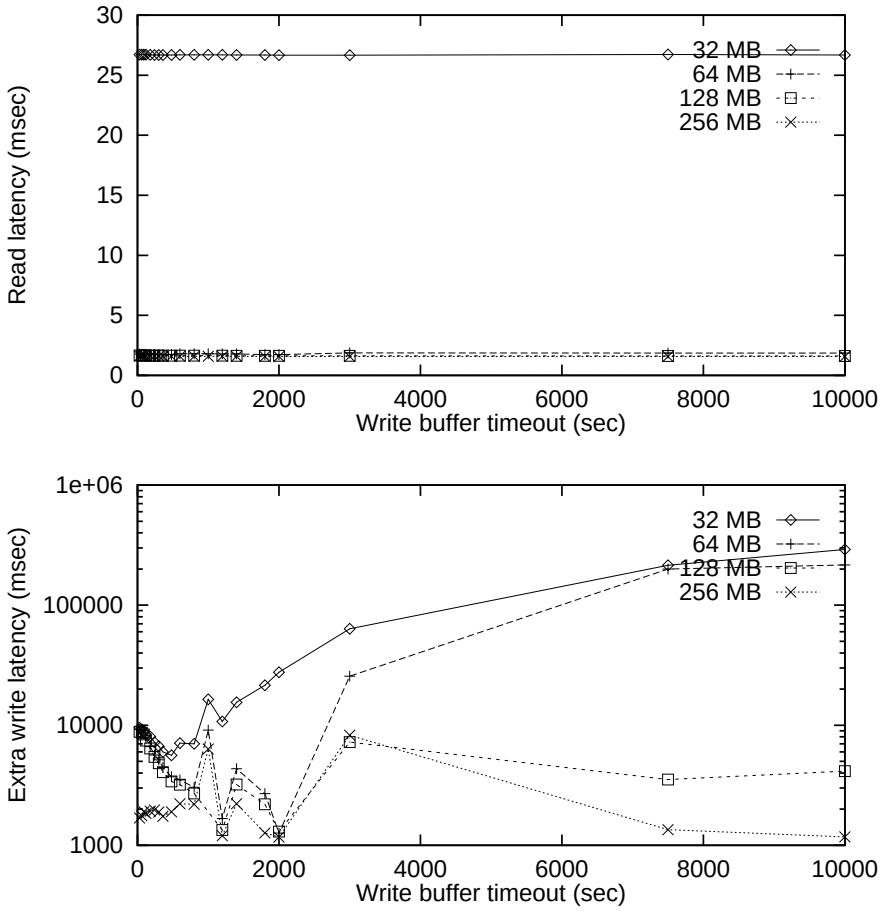


Figure 10. ***Trace 1***

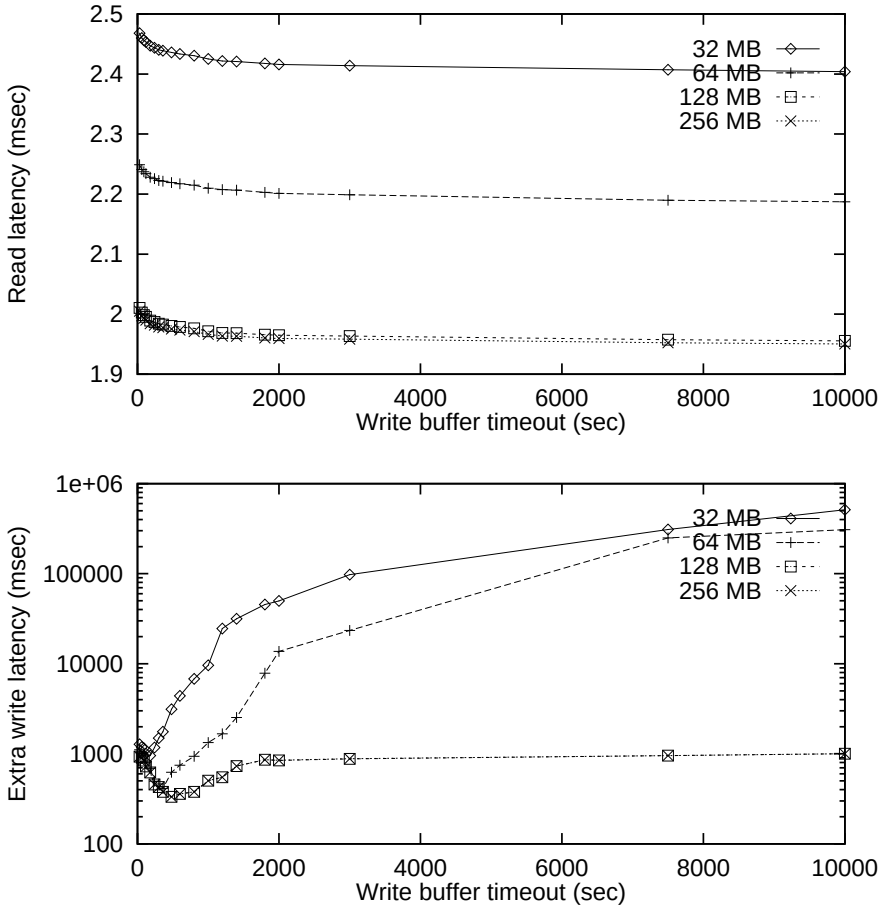


Figure 11. *Trace 2*

severe for the 32 megabyte cache since large write operations are continuously interrupted (in absolute terms, not when compared to the static write buffer timeout). Every interrupt results in at least one extra disk seek. For larger cache sizes write delays are minimal compared to the static write buffer timeout (almost nothing, if data is buffered for 2,000 seconds for large caches). When the read delays are compared with those in Figure 5 a substantial improvement is shown for the case where there is a high disk read/write contention. It shows that the extra write latency causes the write buffer timeout to be changed dynamically to higher values: writes are stalled until disk read/write contention is reduced. The extra delay at $t = 3,000(sec)$ in the non-optimized case, is compensated for by the extra write delay at $t = 3,000(sec)$. When there is hardly any read/write contention, reordering the requests at the disk queue evidently does not help much (and does not cost much).

Trace 2 and trace 3

Trace 2 and 3 do not really show an optimization since there is hardly any contention on the disk when reasonably sized caches are used. In trace 2, latencies are already low. Only if data is forced to disk under the 25% rule (the cache is too small), read latencies improve by this optimization. In trace 3, read latencies are caused by read/read contention, so only larger caches help. For small caches in trace 2, read latency improve at the cost of increased write latencies, which reduce read cache

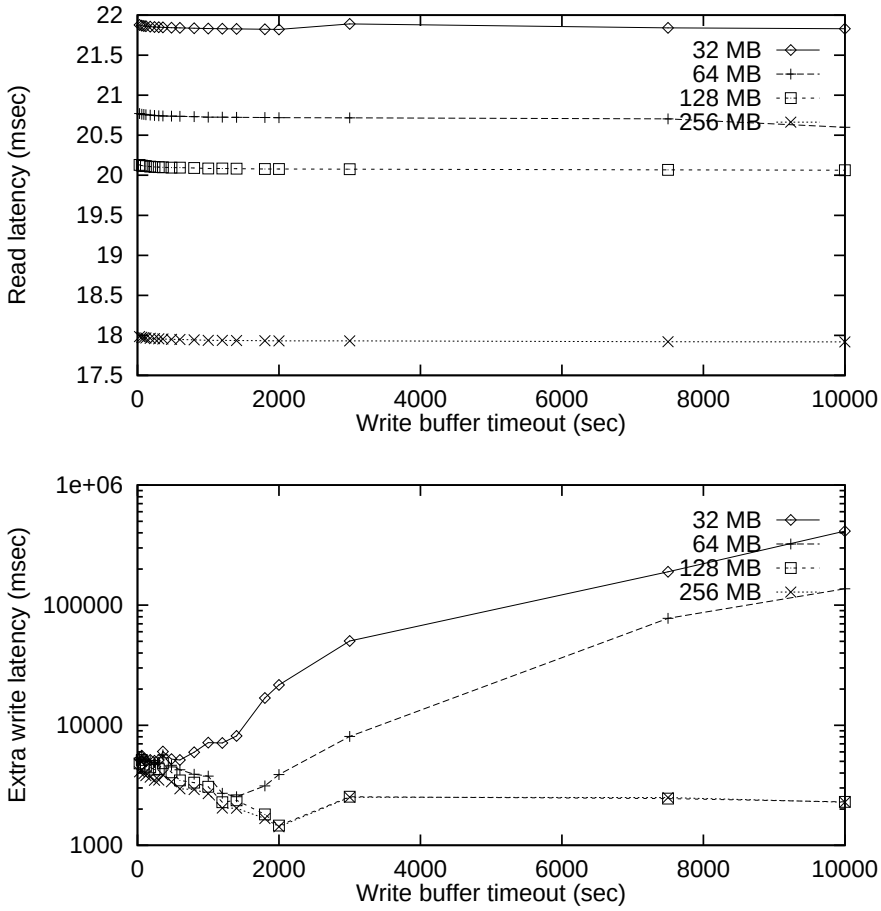


Figure 12. ***Trace 3***

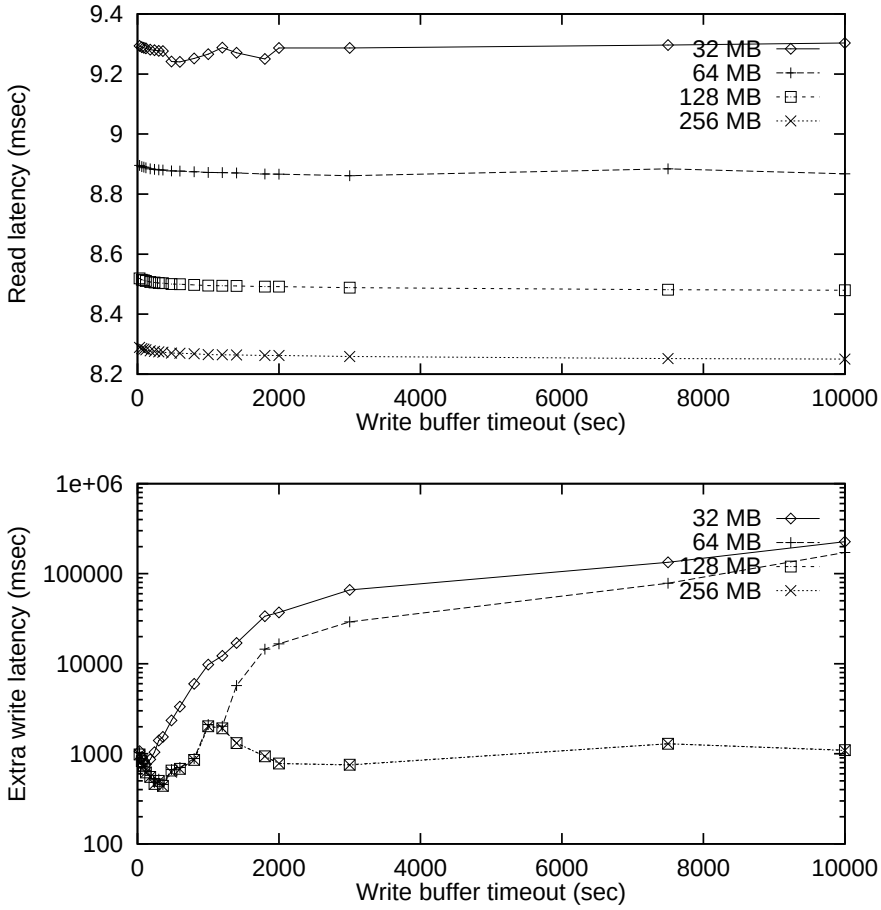


Figure 13. ***Trace 4***

hit rates. The extra write delay, as shown in Figure 11 and Figure 12 are not really dynamically changed (less than 5% of the static write buffer timeout).

Trace 4

In trace 4, read latencies are reduced only if data is forced to disk under the 25% rule. Read latencies are low, and write latencies are only increased by 1–15 seconds, if data is only buffered for at most 1,500 seconds. Although read cache hit rates reduce somewhat by the extra delay, the average read latency decrease is obvious. For this trace, disk queue reordering may help reducing read latencies when the cache is too small.

Trace 5

Again, in trace 5 read latencies are reduced if the queue is reordered and data is forced to disk due to the 25% rule. The difference with trace 4 is that write delays are much larger: 3–100 seconds if data is buffered at most 1,500 seconds. This trace shows that disk queue reordering causes the write buffer timeout dynamically changed to higher values for smaller caches. Disk queue reordering can influence read cache hit rates causing more bytes to be read from disk. It is not clear if disk queue reordering helps in this case. In such a situation, it is an indication that the cache is too small.

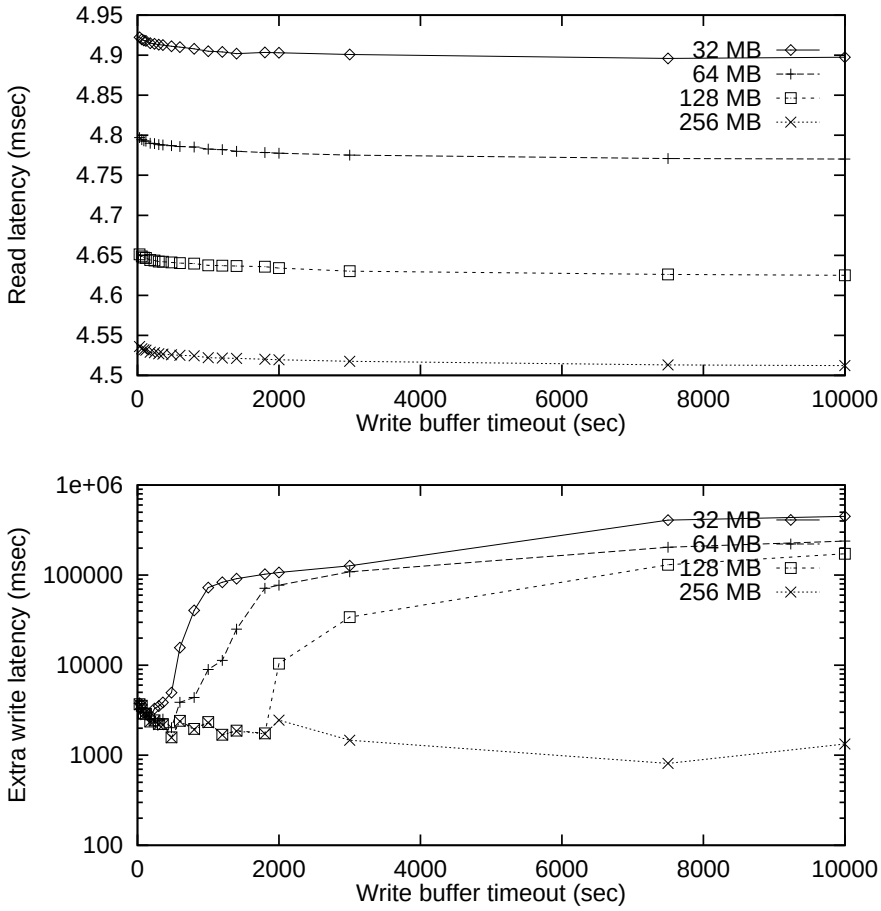


Figure 14. ***Trace 5***

Summary of the traces

Disk queue reordering reduce average read latencies at the cost of delayed writes. In most traces, this extra write delay is less than 5% of the static write delay for higher timeout values and read latencies are minimal. In the simulation it is assumed that if data has been sent to the disk, it is immediately non-dirty (even if it is queued for some time in the disk). In reality, new data will be buffered in ordinary cache. If the extra write delay is long, read cache hit rates may seriously reduce and more bytes need to be read from disk. This effect is not measured in the above described simulation and will in reality increase read latencies. Disk queue reordering basically shows that the static write buffer timeout is dynamically increased. This algorithm may only benificial if there is read/write contention and there is enough cache space. If there is not enough cache space, reordering the queue may be counterproductive since more bytes need to be read from disk due to lower cache read hit rates.

3.3 Summary of the analysis

This section has analyzed the effects of extensive write caching in client and server memory before data is written to disk. It has been shown that memory requirements are reasonable for the first 1,000 seconds. Sprite reported that a block sits 48 minutes in the cache without any client referencing the block before it is replaced by another block. This means that buffering for less than 1,000 seconds in memory does not cost any more memory. Buffering 1,000 seconds in memory reduces the amount of data that needs to be written to disk importantly. The ratio disk reads/writes is reduced by 75–50%, which means that read latencies are less affected by write operations. It even means that in some cases the file system becomes read dominant.

To reduce read latencies even more, a disk queue reordering algorithm is examined. This reordering (reads take precedence over writes) reduces read latencies. However, reordering increase write latencies and reduce cache read hit rates if there is not enough cache space available. It is not clear if disk queue reordering reduces average read latencies in that case.

4 Pegasus File Server Caches

It has been shown in Section 3 that buffering new data in memory for a longer period reduces disk contention and therefore file system latencies. If new data is buffered for longer periods of time in memory, chances are more likely that data is lost if there is a failure in the file system. This section describes a protocol that guarantees persistency of data, even if there are failures in the system by replicating new data over multiple independent nodes.

Ideally, to keep file read and write latencies low, all operations on files are executed locally on the client machine. In reality, some file system operations need to be executed on the remote file server. If a file is not shared, which is the case for most files, all operations are executed locally without having to contact the remote file server. Only if data is not in the client cache, it is retrieved over the network from the remote file server. Data is transmitted to the remote file server when files are closed to guar-

antee data persistency over file system failures[7]. The server notifies the client when it finally writes the data to disk. To allow data sharing, the server negotiates between client machines to pretend to the client applications that they are working on a single site. This roughly describes PFS caching.

People like to share data. When data is shared in a distributed cache, data may become inconsistent if a file is read/write shared. Although not many files are read/write shared, it is still confusing if this happens. To prevent this, caches are synchronized with each other.

PFS caching consists of the following building blocks:

- A cache consistency protocol much like Echo leases and MFS tokens. Client caches and server cache are synchronized by the use of tokens. If a client cache holds a token for a file, it can operate on the file associated with the token, locally. Tokens are created by the central server and delegated to the client cache for a predetermined amount of time to enable progress even when client machines fail, or when the network partitions. The server is able to call back to the client cache to get a token back from the client. This happens when a second client wants to perform a conflicting operation on the file. The token protocol is explained in detail in Section 4.2.

- A data persistency protocol that uses the results which are described in Section 3. New data is buffered for 1,000 seconds in memory to discard as many new bytes as possible in client and server caches. To protect new data against file system failures or network partitions, this data is replicated from the client machine to the server machine when a file that has been written, is closed. The persistency protocol makes sure that the client detects a server failure and the server detects a client failure within a bounded time. Further, through the persistency protocol, the server informs the client when it can guarantee data persistency over file system failures. The persistency protocol is explained in detail in Section 4.3.

- An ordinary client cache, integrated into a UNIX client machine. The client cache currently is a user mode process. The client cache itself is an ordinary block based cache that employs a simple LRU strategy.

- A server cache which is integrated in PFS. The server cache has a thread of control that examines the cache every 5 seconds. If data has been buffered in memory between 360–1,000 seconds, it is flushed to disk. If the cache is full, and the server needs to remove files from the cache, it uses a 25% rule: at most 25% of the available memory is used for write caching.

Both client and server cache are straightforward implementations and do not require further examination.

Communication between client caches and servers is based on unreliable virtual circuits. A virtual circuit is established when the client boots. Messages between client and server are sequenced to detect message loss. It is assumed that messages are delivered in the correct order, network end-to-end latency is low and bandwidth is

[7] This strategy may be too *heavyweight*, write latencies are increased because of this transmission. A flag may be introduced to control the CLOSE(2) semantics.

high. It is also assumed that error rates are low $\ll 0.1\%$. Since PFS uses a high speed ATM network [McAuly89], all assumptions are fair.

On top of this virtual circuit, two types of communications take place: an RPC protocol and non-acknowledged message transfer. The RPC protocol [Birrell83] is used by the client caches to obtain and yield tokens, and to send and receive data. RPC messages are acknowledged by the server some time later when the requested operation has been executed. The acknowledge holds the sequence number which has been sent as part of the RPC request to bind response messages to the original request.

Non-acknowledged messages are also transmitted along the same path. Non-acknowledged messages are keep-alive messages, a token revoke and a data persistent message. The non-acknowledged messages are only used to notify the other party of some event.

The moment a message (RPC or non-acknowledged) is lost, which is detected by client or server, the connection is terminated. The client needs to setup a new connection in this case and needs to re-establish its state. Every ε seconds both client and server exchange at least one message. If there is no traffic, the client and server exchange a keep-alive message. Failure to receive such a message (in $\varepsilon + \delta$ seconds) causes the virtual circuit to be terminated by the party that detects the message loss. This means that within a bounded time, both server and client know that there is a failure on the route to the other party. Typical values for ε and δ are 30 and 5 seconds.

Section 4.2 describes the token protocol, and Section 4.3 describes the persistency protocol. Section 4.1 describes the invariants and assumptions for the Pegasus file server. Since [Bosch93] does not state these invariants and assumptions explicitly, they are repeated and formalized here.

4.1 Invariants and Assumptions

This subsection states the invariants of the PFS architecture and the assumptions made when the file system was designed. It also describes why it is useful to the file server design and which parts of the file system it influences.

Assumption 1 *PFS has been designed such that there is no single point of failure which can cause data loss in the system. A failure may only cause temporary unavailability.*

Assumption 1 states that all modules in the system have been designed such that a single failure does not influence the persistency of data. Volatile data replication to multiple independent nodes is used to protect new data. When a node fails, the backup node can provide the data. In case of a failure, the backup node makes the volatile data persistent. failures.

Assumption 2 *PFS assumes that while a failure is being repaired, no second failure will happen.*

Although assumption 1 and assumption 2 makes the system simpler, it is by no means a statement that two or more failures will not happen at the same time. If two independent failures occur but the nodes that fail are not depending on each other, assumption 1 still holds. The chances that two nodes fail that are depending on each other are considered zero. In reality, they are not and backup facilities are still required. By assuming that no second failure will take place while the first failure is

being repaired, the file server design does not have to account for complex recovery mechanisms. A global power failure is considered a single point of failure; it is guaranteed that the file server itself will continue running since it is powered by a UPS.

Invariant 1 *A PFS file behaves as a regular register [Lamport85] with respect to individual reads and writes.*

Invariant 1 states that file updates to stale files do not occur since a file is always consistent. The invariant does not imply a file request ordering. Applications which need a strict ordering of updates or want to use a complete file as a regular register still need to lock files since it is impossible that a file can be cached wholly in memory. PFS behaves as if all clients and server are running on the same machine: it provides *single site semantics*.

4.2 Token Management

In PFS, the server holds tokens. A token is a representative for a file. Tokens are delegated to client caches when the client cache asks for a token on behalf of a client application. If a client cache *owns* a token for a file after it has been delegated by the server, the client cache performs the operations on the file locally. Two types of tokens exist: shared tokens to read a file, and exclusive tokens to read and write a file. Many client caches can read a file, but only one client can write to a file at the same time. Multiplexing the token to applications at the same client machine is the responsibility of the client cache.

Tokens have a *life time*: τ. When a token is alive, the client cache serves client application requests for the file. Tokens are automatically refreshed every time the server receives a message from the client cache. If the server has not heard from the client cache when the life time expires, the server considers the client cache *dead* and automatically revokes the token from the client cache. This means that progress is possible even when client machines fail or when the network partitions. A typical value for τ is 30 seconds.

If the server revokes the token from the client, the client is guaranteed to have at least $\tau - \varepsilon$ seconds to flush dirty data to the server when there is no failure in the system. τ and ε have been chosen such that the client is able to send at least the largest possible file to the server. If there are failures in the system, things get more complicated:

1. The client owns a read token and the connection to the server breaks. The client discards the token, and when the connection has been re-established, the client obtains a new read token for the file. Note that the client does not remove the cached file from its cache: only the token is retrieved again from the server. When a token is granted, the server also sends the time of last modification to the client cache. The client cache can then decide if the cached copy is out of date.

2. The client owns a write token, the connection breaks and the client has not modified the file. In this case, the token is discarded, and when the connection to the server is re-established, the client obtains a new write token from the server.

3. The client owns a write token, has modified the file, and has not flushed the updates to the server. If the client crashes, it looses all state and the server retrieves the token when the lifetime expires (or when the server detects that the client machine has rebooted).

4. The client owns a write token, has modified the file, has not sent the changes to the server and the server crashes. The server guarantees that recovery time is longer than $\varepsilon + \delta$ seconds. The client detects the failure, and retrieves a new token when the connection to the server has been re-established. Once the token is acquired, another client may have modified the file (if the server did not crash, but there was a network partition). In that case, the changes are applied to the file again. This does not violate the semantics of regular registers since the changes were not made permanent and the two writes were overlapping.

5. The client owns a write token, has modified the file, has not made the changes permanent and the server has sent a revoke token message that was lost. This case is considered a network failure and is detected by the client cache by unsynchronized sequence number on receipt of the next keep-alive message. The network connection is terminated, and the server revokes the token.

Obtaining and releasing tokens

Clients request a token by sending a token request from the server. The server maintains all requests in a strict FIFO order which guarantees that all requests will be served eventually. It works as follows:

- If the server token request FIFO for a file is empty, the token request is granted, and the server acknowledges the token request to the client cache. The server records the identity of the client cache which holds the token, and records the expiration time.

- If the server token request queue for a file is non-empty, and the outstanding tokens are not conflicting with the requested token, the token request is granted and the server acknowledges the token request to the client cache. This immediately increases the expiration time of the outstanding tokens to the new expiration time. The file server records the identity of the client cache.

- If the server token request queue for a file is non-empty and the token request is conflicting with the outstanding tokens, the file server calls back to the client caches which hold the tokens and asks those client caches to release the tokens. Whether the client caches holding the token release the requested token is up to those client caches. If the client caches do not release the tokens, the tokens will expire eventually. If the client caches release the token, they send a yield token message to the server. Further file updates require the client cache to obtain a new token. When all outstanding tokens have been released or timed out, the token request is granted.

If the connection breaks, the server clears all requests for that client. Further, it discards all tokens delegated to the client. This means that the client needs to reacquire all tokens it has requested or which were delegated by the server.

Generally, clients do not release tokens voluntarily. Only if the server asks the client to release a token, the token is yielded. This means that in general files can be served from the client cache if the file is already in the client.

Sending and receiving data

If a client cache holds a token for a file, the client cache can read and/or write to a file. The client cache performs RPCs to the server to read or write a part of the file. The requests fail if the token life time has expired on the file.

4.3 Persistency Protocol

The persistency protocol is responsible for ensuring the persistency of data once the data has been sent to the file server. Since data is kept in memory much longer than in traditional UNIX systems, chances are more likely that data is lost when there is a failure in the system. In the first prototype implementation, data is transmitted to the file server when the file is closed. The server stashes the received updates in the server cache and starts a timer for the data. When the data is between 360 and 1,000 seconds old, the data is sent to disk and a message is sent to the client cache that originally created the data. A message is sent as well when another client updates the file while the data is stored in the stash.

Several complications are possible:

- While the data is in the server stash, the server fails and looses all state. Since keep-alive messages are exchanged every ε seconds, the client cache detects a server failure and makes a backup of the data on the local disk (if it has one), picks another file server to store the data temporarily (if there is one), persuades another client to keep a copy of the data (if this is supported) or simply prints a message on console informing the user that persistency cannot be guaranteed if the client machine fails.

- The client cache crashes. In this case the server detects the client failure after ε seconds and starts writing the data to disk without further delay.

- The network partitions. Both client and server decide in ε seconds that the other party died. The server immediately writes all updates from that client to disk. The moment the client re-establishes the connection, it checks the last update time to the file by re-acquiring a token. If the update time is less than the client cache's copy, the server crashed and the updates are retransmitted to the server. If the update time is equal to the client cache's copy, the network had partitioned and the changes are safe. If the update time is higher, another client has made an update to the file and the server was not able to notify the former client that its data safe, or the server has lost the former client's update. A warning will be printed on the user's console and a backup file is written which holds the conflicting data. There is no way to prevent this from happening in a synchronous system [Fischer85].

4.4 Summary

A simple mechanism has been presented to organize client caches and server cache into a distributed caching scheme that provides single site semantics. Failures in the system are detected within a bounded amount of time since the system is completely synchronous. A recovery protocol is provided to guarantee new data persistency over system failures. Liveliness is guaranteed by delegating the responsibility for a file to the client cache only for a predetermined amount of time.

The system is not yet fully built. Before something can be said about performance, it needs to be used as standard file system for a department.

5 Conclusions

Low latency file systems for a UNIX workload can only be designed by carefully examining the access patterns of the workload. UNIX workload is characterized by a moderate persistency guarantees, low program execution times and high overwrite factors. The latter can be used to decrease the read/write contention on the disk. By postponing disk write operations, less data is written to disk, which reduces disk read/write contention and read latencies.

Disk queue re-ordering to give precedence to read operations at the disk may only help if the cache is too small or when the static write buffer timeout was chosen badly. Under some circumstances, read latencies are reduced by a temporary write buffer timeout increase. This only works if there is enough cache space. It is not yet clear if the extended write delays influence read latencies through lowered read cache hit rates. This happens when the cache is too small for the workload.

Buffering new data in (volatile) memory for longer periods does have the disadvantage that more data can be lost if there is a system failure. A simple protocol has been designed to replicate new data over a distributed cache to prevent data loss when there is a failure in the system.

Acknowledgements

First of all, I would like to thank (in alphabetic order) Richard Golding, Arne Helme, Jack Jansen, Sape and Sjoerd Mullender, Guido van Rossum, Paul Sijben, and Tage Stabell-Kulø for their support, criticism and remarks during this work and for reading the various drafts. Without their help, I would never have finished this work. Further, I would like to thank Richard Earnshaw for providing helpful comments when I was doing the analysis, and the other Pegasi for their remarks, and support during this work. Also I would like to thank CWI for sending and receiving packets to and from Twente. These packets carried many versions of this thesis. Last but not least, I would like to thank Fred Gansevles for providing many CPU cycles which I needed for the simulations.

Als Jeannette me niet af en toe meegesleept had naar Vertigo had ik dit werk nooit kunnen voltooien.

References

[Baker91] Mary G. Baker, John H. Hartman an Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove CA (USA)), volume 25, number 5 of Operating Systems Review, pages 198–212, October 1991.

[Barham94] P. R. Barham, M. D. Hayter, D. R. McAuley, and I. A. Pratt. Devices on the Desk Area Network. *submitted to Journal on Selected Areas in Communications*, 1994.

[Birrell83] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *Proceedings of 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, New Hampshire). Published as *Operating Systems Review*, **17**(5), October 1983.

[Bosch93] Peter Bosch, Sape Mullender, and Tage Stabell-Kulø, *Huygens File Service and Storage Architecture*, *Esprit BROADCAST workshop* (Newcastle, UK). Published as Nick Cook, editor, *BROADCAST Technical Report Series, 1st year report*, **3**. University of Newcastle, OCT 1993.

[Burrows88] Michael Burrows. *Efficient Data Sharing*. PhD thesis, published as Technical report 153. University of Cambridge, DEC 1988.

[Digital92] Digital Equipment Corporation. *DSP3105 Specification*. Digital Equipment Corporation, OCT 1992.

[Fischer85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, **32**(2):374–82, April 1985.

[Hartman93a] John H. Hartman. *Using the Sprite file system traces*, Version 1.0. University of California, Berkeley, CA 94720, USA, may 1993.

[Hartman93b] John H. Hartman and John K. Ousterhout. Letter to the editor. *Operating Systems Review*, **27**(1):7–10. Association for Computing Machinery SIGOPS, January 1993.

[Hyden94] Eoin Hyden. *Operating System Support for Quality of Service*. PhD thesis. University of Cambridge, February 1994.

[Kleiman86] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. *1986 Summer USENIX Technical Conference* (Atlanta, GA, June 1986), pages 238–47. USENIX, June 1986.

[Lamport85] Leslie Lamport. On interprocess communication. Technical report 8. Digital Equipment Corporation, Systems Research Center, Palo Alto, Ca, December 1985.

[Mann93] Timothy Mann, Andrew Birell, Andy Hisgen, Charles Jerian, and Garret Swart. A Coherent Distributed File Cache With Directory Write Behind. Technical report 103. DEC SRC, June 1993.

[McAuly89] Derek Robert McAuley. *Protocol Design for High Speed Networks*. PhD thesis, published as Technical report 186. University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, January 1990.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[Mullender89] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba — high-performance distributed computing. Technical report CS–R8937. Centrum voor Wiskunde en Informatica (Centre for Mathematics and Computer Science), Amsterdam, August 1989.

[Mullender94] Sape J. Mullender, Ian M. Leslie, and Derek McAuley, *Operating System Support for Distributed Multimedia*, *Usenix 1994 Summer Conference* (Boston). Usenix, June 1994.

[Ousterhout85] John Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. *Proceedings of the 10th Symposium on Operating System Principles*, pages 15–24, December 1985.

[Ousterhout88] John K. Ousterhout, Andrew R. Cherenton, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, February 1988.

[Patterson88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of ACM SIGMOD Conference*, pages 109–16, June 1988.

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *Proceedings of Summer UKUUG Conference*, pages 1–9, July 1990.

[Popek85] Gerald Journal Popek and Bruce Journal Walker, editors. *The LOCUS distributed system architecture*, Computer Systems Series. MIT Press, Cambridge, Mass, 1985.

[Pratt93] I. Pratt. *ATM camera V1*, ATM Document Collection 2 (The Orange Book). University of Cambridge, February 1993.

[Ritchie74] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, **17**(7):365–75, July 1974.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove CA (USA)), volume 25, number 5 of Operating Systems Review, pages 1–15, October 1991.

[Sandberg85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Association Summer Conference Proceedings of 1985* (11-14 June 1985, Portland, OR), pages 119–30. USENIX Association, El Cerrito, CA, 1985.

[Schoch82] J. F. Schoch, Y. K. Dalal, D. D. Redell, and R. C. Crane. Evolution of the Ethernet local computer network. *IEEE Computer*, **15**(6):10–28, August 1982.

[Schroeder85] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island WA (USA)). Published as *Operating Systems Review*, **19**(5):25–34, December 1985.

[Seltzer92] Margo Ilene Seltzer. *File system performance and transaction support*. PhD thesis. University of California, 1992.