

On Providing an Efficient and Reliable Virtual Block Storage Service

EBEN ESTERHUYSE



THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMMERCE
AT THE UNIVERSITY OF STELLENBOSCH

Promoter: P.J.A. de Villiers
November 2000

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature:

Date:

Summary

This thesis describes the design and implementation of a data storage service. Many clients can be served simultaneously in an environment where processes execute on different physical machines and communicate via message passing primitives. The service is provided by two separate servers: one that functions at the disk block level and another that maintains files.

A prototype system was developed first in the form of a simple file store. The prototype served two purposes: (1) it extended the single-user Oberon system to create a multi-user system suitable to support group work in laboratories, and (2) it provided a system that could be measured to obtain useful data to design the final system. Clients access the service from Oberon workstations. The Oberon file system (known as the Ceres file system) normally stores files on a local disk. This system was modified to store files on a remote Unix machine. Heavily used files are cached to improve the efficiency of the system.

In the final version of the system disk blocks are cached, not entire files. In this way the disks used to store the data are unified and presented as a separate virtual block service to be used by file systems running on client workstations. The virtual block server runs on a separate machine and is accessed via a network. The simplicity of the block server is appealing and should in itself improve reliability. The main concern is efficiency and the goal of the project was to determine whether such a design can be made efficient enough to serve its purpose.

Opsomming

Hierdie tesis omskryf die ontwerp en implementasie van 'n data stoor diens. Verskeie gebruikers word bedien deur die diens wat funksioneer in 'n verspreide omgewing: 'n omgewing waar prosesse uitvoer op verskillende masjiene en met mekaar kommunikeer met behulp van boodskappe wat rondgestuur word. Die diens word verskaf deur twee bedieners: die eerste wat funksioneer op 'n blok vlak en die ander wat lêfs onderhou.

'n Prototipe lêr diens is ontwikkel deur middel van 'n basiese lêr stoor. Die prototipe het twee funksies verrig: (1) die enkel gebruiker Oberon stelsel is uitgebrei na 'n veelvoudige gebruiker stelsel bruikbaar vir groepwerk in 'n laboratorium omgewing, en (2) 'n stelsel is verskaf wat betroubare en akkurate data kon verskaf vir die ontwerp van die finale stelsel. Oberon werkstasies word gebruik met die lêr diens. Die Oberon lêr stelsel (ook bekend as die Ceres lêr stelsel) stoor normaalweg lêrs op 'n lokale skyf. Hierdie bestaande stelsel is verander om lêrs te stoor op 'n eksterne Unix masjien. Lêrs wat die meeste in gebruik is word in geheue aangehou vir effektiwiteits redes.

Die finale weergawe van die stelsel berg skyf blokke in geheue, nie lêrs nie. Hierdie metode laat dit toe om data te stoor op 'n standaard metode, bruikbaar deur verskillende tipes lêr stelsels wat uitvoer op verskeie gebruikers se werkstasies. Die virtuele blok stoor voer uit op 'n aparte masjien en is bereikbaar via 'n netwerk. Die eenvoudige ontwerp van die diens is opsigself aanloklik en behoort betroubaarheid te verbeter. Die hoof bekommernis is effektiwiteit en die hoofdoel van die projek was om te bepaal of hierdie ontwerp effektief genoeg gemaak kon word.

Acknowledgements

I wish to thank the following people who have assisted me during my studies, in particular

1. Dr. P.J.A. de Villiers and Prof. AE Krzesinski
2. The members of the Hybrid project
3. The Departement of Computer Science at Stellenbosch
4. My parents, close friends and girlfriend Sanette Ludick
5. The Foundation for Research Development (FRD) and
6. the University of Stellenbosch for financial assistance.

Contents

1	Introduction	1
2	A Prototype File Service	3
2.1	The Runtime Environment	3
2.2	File Service Overview	5
2.3	Caching Environment	8
2.4	Directory Environment	10
2.5	Cache Replacement Policy	11
2.6	Physical Disk Storage Service	14
3	A Block Storage Service	16
3.1	Design Overview	16
3.1.1	Stateless Service	17
3.1.2	Blocks	19
3.1.3	Partition Table	20
3.2	Efficiency	22
3.2.1	Block Cache	23
3.2.2	Bitmap Cache	25

3.2.3	Least Recently Used (LRU) vs. Least Frequently Used (LFU) . . .	28
3.3	Reliability and Availability	31
3.3.1	Reliable Service with Cache Coherency	31
3.3.2	Reliability with Stable Storage	32
3.3.3	Storage Cleanup	34
3.3.4	Service Availability	35
3.4	Service Functionality	37
3.5	Summary	43
4	Performance Comparison	45
4.1	Trace Collection	45
4.2	Miss Ratio Analysis	46
4.2.1	Comparison of Different Age Factors	46
4.2.2	Comparison of CFS and BFS	47
4.3	Service Analysis	49
4.3.1	Bandwidth	49
4.3.2	Response Time	50
5	Conclusion	55
A	CFS Service Commands	59
B	Disk Storage Service (TcpSVR) Commands	61
	Bibliography	62

List of Tables

1	BFS Average Bandwidth Measured in Kilobytes per second	50
2	Average Response Time Percentage Difference Calculated under different User Loads	53
3	CFS File System Operation Table	59
4	CFS User Operation Table	60
5	TcpSVR Operation Table	61

List of Figures

1	CFS File Service Layout	6
2	File Cache Lookup Table	8
3	Directory Entry Lookup Table	11
4	Block Description	20
5	Block Cache Layout	26
6	Virtual Block Storage Service Environment	38
7	Virtual Block Storage Service Header Type	39
8	Block Storage Service Module Layout	42
9	Read Miss Ratio Analysis for different LFU Age Factors	47
10	Read Miss Ratio Analysis of CFS and BFS.	48
11	Response Time of CFS and BFS for Reads	51
12	Response Time of CFS and BFS for Writes	52

Chapter 1

Introduction

Many institutions use dedicated server machines to provide a file service to client workstations. A file server manages files — named objects containing data. Files can be created, read from, overwritten or removed. Various attributes (e.g. creation date, length etc.) are stored for each file. Files and their attributes are usually stored on the same physical storage medium (e.g. disks or tapes).

File Server

In a distributed environment numerous clients may communicate with one or more file servers via a network. File servers are reactive programs that allow access to physical storage media. A reactive system is a program that reacts continually to requests coming from its environment.

Problem
Description

The goal of this project was to design a file service where the following issues were paramount:

Design goals

- Fast response time — a primary concern in an interactive environment.
- Reliability — the data on the physical medium should always remain in a consistent state in the face of physical failures of storage media.
- Availability — re-connection should be established within seconds after any interruption of service.

A subsequent goal was to implement the management of disk blocks and the management of files as two completely separate servers. This simplifies the design substantially.

In addition, different types of file systems can communicate with the same virtual block storage server, which improves the flexibility of the service. Whether such a service would be efficient enough was to be seen.

The design and implementation of a virtual block storage server is described in Chapter 3 with special emphasis on efficiency, reliability and availability. The basic design criterium is flexibility. Client workstations are powerful enough to manage structural information about files. As a result, it is possible to design a flexible service that caters for different types of file systems.

A Virtual Block
Storage Server

A prototype file service was developed first. Measurements obtained from the prototype were used to design the final version of the system. Chapter 2 introduces the runtime environment in which both these services function as well as the design methodology behind the prototype file service.

A File Service Pro-
totype

Results obtained from testing the final version of the system with data collected from the prototype are discussed in Chapter 4. The approach of separating the issues of disk block storage from file storage proved surprisingly effective. Conclusions in this regard are discussed in Chapter 5.

Conclusion

Chapter 2

A Prototype File Service

This chapter presents a prototype file service developed for the environment described in Section 2.1. Instead of designing another file system, an existing one was extended to provide a required, fully functional file server from which accurate trace information was collected. Section 2.2 presents the design methodology of this prototype and basic service layout. The caching environment is presented in Section 2.3, while Section 2.4 describes the maintenance of user directory entries. Section 2.5 describes the cache replacement policy used by the prototype file service. The physical storage environment is presented in Section 2.6.

Chapter Overview

2.1 The Runtime Environment

Reactive systems, like file servers, typically function in an environment closely coupled to specialized hardware. To support the file server discussed here, a microkernel is used. The microkernel provides some basic functionality: memory- and process management, interprocess communication (IPC) and low-level peripheral device drivers [MdV94]. The Gneiss microkernel was designed to serve as a platform for distributed computing [TvR85] and the development of reactive systems [Mul94].

Microkernel

The Gneiss microkernel supports *Virtual Machines* (VM). VMs are designed to support client-server applications. A VM is similar to a process in commercially available operating systems like Unix and Microsoft NT. VMs are scheduled by the kernel and each VM has an associated protected address space. A VM supports one or more threads of

Virtual Machine

execution. Threads are also scheduled by the Gneiss kernel. VMs cooperate with each other via synchronous message-passing primitives (interprocess communication) supported by the microkernel. A time quantum is assigned to a VM when it is activated. Priorities are assigned to VMs and the VM with highest priority will be activated when the time quantum of the currently executing VM expires. Threads are allowed to run until a kernel call is encountered or until a hardware interrupt occurs. If a hardware interrupt causes a new VM to be activated (because it has a higher priority than the currently active VM), the thread that was interrupted is guaranteed to be activated again until it reaches a kernel call. This policy eliminates the need for explicit synchronization primitives to protect data shared among threads supported by the same VM.

Remote communication between VMs on different physical machines is based on the Versatile Message Transfer Protocol (VMTP). VMTP is a lightweight transaction-based protocol designed specifically for distributed systems [Che89, Che88].¹ A global name server enables VMs in the distributed environment to contact remote VMs. Remote server VMs register a port identifier (see p 25 of [Mul94]) which includes their Internet Address (IP). Client VMs obtain the port identifier from the name server and communication can commence between client and server. A special VM is used to support a TCP stack [Pos81b] while the IP protocol [DH98, Pos81a] is implemented inside the microkernel.

Remote IPC

The Oberon system [WG92], both a programming language and an operating system, is used to develop applications for the Gneiss microkernel. The Oberon system consists of a hierarchy of modules with clearly defined interfaces which are contained inside a single VM. No explicit boundary is defined between system modules and user defined modules. A graphical viewer system [FM98] provides a productive working environment. In addition to a graphical user interface (GUI) the Oberon system provides tools for program development like a compiler [Cre94] and a static linker [Wet98]. Reactive systems for the distributed environment are implemented in Oberon by using the compiler and static linker.

Oberon System

The file system provided by Oberon is known as the Ceres file system [WG92]. It is a simple, flat (non-hierarchical) file system based on traditional techniques. However,

Ceres File System

¹The current VMTP implementation bandwidth is more or less 700K per second, tested under a light network load on a 10-Mbit Ethernet network, using a Pentium-166 with a Western Digital 8003 network card as the server machine and a Pentium-100 as the client machine.

it has one peculiar property that influenced the designs discussed in this thesis: a new copy of an entire file is stored on disk each time it is updated. Older copies of files remain on disk and must be removed during a garbage collection phase. To recover disk space occupied by old copies of files, a garbage collection process is used that is conceptually similar to memory garbage collection [JL97]. The single-user Ceres file system was adapted to a multi-client distributed environment.

2.2 File Service Overview

A first prototype of a file service was designed to provide an efficient file service for clients working in the Oberon environment. It is well known that the performance of a file service can be improved significantly by maintaining a cache in memory. Makaroff et al. [ME90] demonstrated the improvement in performance when using a disk cache in a distributed system. As expected, the cache hit ratio improves significantly as the cache size is increased.

Design
Methodology

Blaze et al. [BA92] discuss the benefits obtained when using client caches in a distributed system and the first prototype of our file service CFS was designed with this knowledge in mind. The Oberon system loads executable code into memory one module at a time. A module is loaded and bound dynamically when it is first referenced and it remains in memory. This approach emulates a client cache. Modules can be unloaded explicitly or modules that are no longer needed can be unloaded during garbage collection. Many clients need the same modules, for instance when booting the Oberon system. Consequently, a significant performance improvement can be gained by using a cache.

It has been shown that it can be beneficial to cache files. For example, the Andrew file system [LS90] caches entire files. When a file is opened the entire file is loaded into the cache to make all subsequent reads and writes more efficient. Amoeba's Bullet file server [DOKT91] also uses file caching. The Bullet file server implements an immutable file store. Three basic operations are provided: *read-file*, *write-file* and *delete-file*. After a file has been created it can only be read or deleted. To modify a file it must be read into memory, updated and written to disk as a new file, the old copy being deleted.

An outline of the first prototype file server, CFS, is shown in Figure 1. The Cache File

Service Description

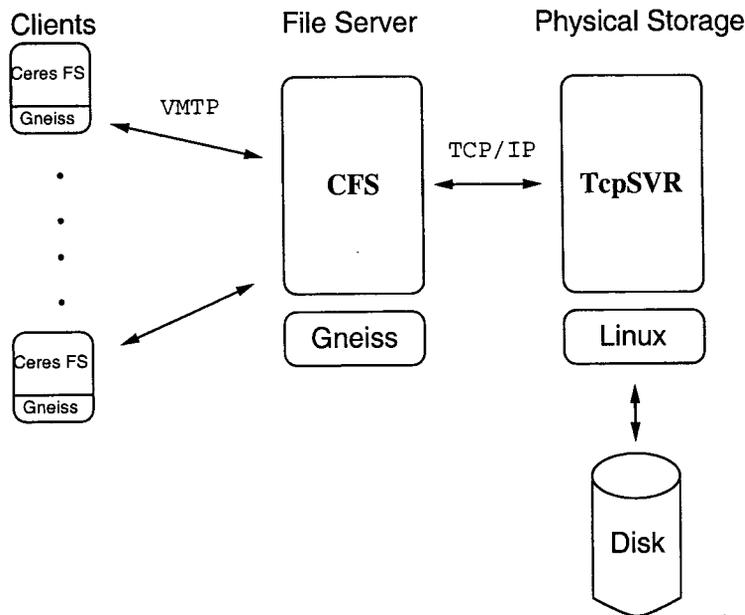


Figure 1: *CFS File Service Layout*. This figure shows two servers running on two different machines. Firstly the CFS server provides a caching facility for files and other user data. Secondly the TcpSVR provides non-volatile storage on a disk. Clients communicate with the service via the Ceres File System, using the VMTP communication protocol, while CFS communicates with the TcpSVR via TCP/IP.

Server (CFS) is a concurrent, multiuser, write-through server that caches entire files and some additional user data such as password information.

A first prototype that could provide a much-needed service was implemented by using the Linux file system to store files. A simple disk storage server (TcpSVR) on Linux is used to interface the standard Linux file system to CFS (running on a separate machine) which supports Oberon clients. The basic reason for not implementing CFS on Linux was that the VMTP protocol was needed for communication between the client machines and the file server and VMTP is not supported by Linux. So it was decided to let the client machines (each containing the Ceres file system) communicate with CFS via VMTP, while CFS communicates with the disk storage server on Linux via TCP/IP.

Clients connecting to CFS must provide a user name. Each user belongs to a group (which is represented as a special username). All groups belong to a system wide special username. Users may read from their own file data area or accounts, the group account they belong to and the system wide user account, while groups may read from

their own accounts and the system wide user account. Users, groups and the system wide user are only permitted to write to their own accounts. A special user, `SysAdm`, maintains a password and group file. Both these files contain the relevant information for all users of the file service. Because no user is permitted to read another user's files, only the `SysAdm` user can read the password and group file. ²

When CFS is started up it creates a TCP connection with the disk storage server on Linux. Once connection is established, the relevant service data structures are initialized. CFS uses four main data structures: a password table, a group table, a file cache and a directory cache. The password and group tables enable efficient user access validation by caching all the relevant user information. At service startup both the password and group files are read into memory from the storage server. Once the group information is available all group directory entries are read into a directory cache (see Section 2.4 for a discussion of the directory cache). The group and system wide users directory entries are maintained in the directory cache until cache space is required (see Section 2.5). The service registers its port identifier at the global name server and creates the client connection threads. Service Startup

On connection to the file server, clients (users) supply a username and password. CFS compares this to the password and group table, and returns a *capability* that is used by the clients for secure access to the file service. A capability is a data structure containing one or more fields of integers that make it difficult to counterfeit. Access to the prototype file service is only attained through the correct capability. The capability of CFS consists of twelve bytes. The first four bytes are reserved for a file identifier assigned by the file service. The following two bytes contain the user identifier (UID) and the next two bytes contain a super user identifier (SUID). The last four bytes are reserved for a random number generated at service startup. To communicate with the file service, a client must supply, when necessary, the correct file address in the cache (file identifier), the correct user identification number (UID and SUID) to which the file is assigned and the correct random number. The SUID field enables a user to read a group or system wide user file and is assigned by the service when searching for the file. In addition to the capability, clients supply other information if necessary (i.e., the filename, file sector number etc.). This provides a level of server security. ³ Service Security

CFS accepts several different operations. Table 3 in Appendix A shows the file system Service Operations

²CFS allows logins based on the file information build at startup time.

³The *UFC-crypt* method is used as password encryption scheme.

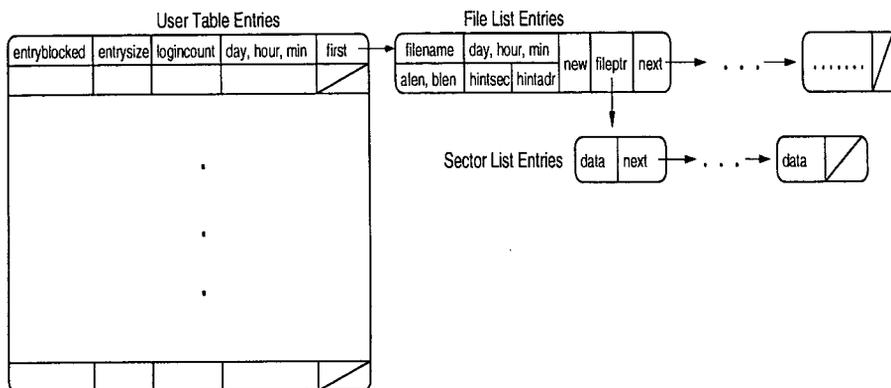


Figure 2: *File Cache Lookup Table*. Each entry in the User Table represents a user with a field (**first**) pointing to the initial file in the users link list of files. Each file in the link list of files contains information concerning the file and a link list of file data sectors (**fileptr**).

operation primitives, while Table 4 in Appendix A shows the user related operation primitives.

In addition to the password and group file structures maintained by CFS, two other main data structures are also used. These are (1) the file cache (presented in Section 2.3) and (2) the directory cache (presented in Section 2.4).

Main Data Structures

2.3 Caching Environment

Files are maintained by CFS in a file lookup table or file cache showed in Figure 2. A file is divided and stored as sectors in the file cache. Each entry in the file lookup table represents a user (UID) and contains fields that (1) provide user level locking (**entrylocked**), (2) the current user file cache space occupation (**entrysize**), (3) the number of users logged on in an entry (**logincount**), (4) the time of last entry access (**day, hour, min**) and (5) a list of cached files (**first**).

Cache Lookup Table

The **entrylocked** field in the cache lookup table enforces single user access to a user's cache space. Although this prohibits users from working in the same table entry, different users (with different UIDs) are still permitted to work simultaneously. When a client tries to access a cache area and another client is busy in the same area, the second client must wait. This prevents two users from working in the same cache area

Synchronization

simultaneously.

Non-group user file entries are removed at logout time to preserve cache space. To prevent removing files of clients who logged in on the same user account, a `logincount` field maintains the current number of clients working in the same user account. When the last client in a user entry logs out, the user's file entries are removed. The `day`, `hour` and `min` fields indicate the last time a specific table entry was accessed and are used to prevent dead login sessions (users who do not logout) from occupying unnecessary cache space as well as assisting the cleanup procedure discussed in Section 2.5. The `first` field in each table entry points to the users link list of file entries. New file entries, created by a client or read via the `TcpSVR`, are inserted in front of the file linked list to enable faster searching for newly inserted files.

Caching Policy

Runtime garbage collection is used to handle various versions of files, as required by the Ceres file system. On file registration (the creation of a copy of a file on disk) to the storage server, the user's cache space is searched for older copies of the file and removed. Files are not removed under certain circumstances, for instance when viewing a file from an Oberon compressed archive. The Oberon `Compress` utility creates a separate temporary file called `temp.temp` for each file viewed from within an archive. Overwriting such a file would result in an inconsistency with the Oberon system when a second file is viewed from within the compressed archive, and the first copy is referenced again.

Runtime Garbage
Collection

Each entry in the link list of files contains the following fields: `filename`, `alen`, `blen`, `day`, `hour`, `min`, `new`, `fileptr`, `hintsec`, `hintadr` and `next`. Besides the `filename`, each file entry in the file link list contains the number of sectors (`alen`) in the link list of sectors (pointed to by `fileptr`) and the size of the last sector (`blen`) of the file in bytes. The time attributes (`day`, `hour` and `min`) are updated with each reference to the file and assist the cleanup procedure (see Section 2.5). A `new` flag indicates the temporary nature of the file, thus showing whether the file has been registered at the storage server. The `next` field points to the next entry in the file link list.

File Link List Entry

To prevent memory fragmentation when large memory blocks are allocated and later deallocated, file data is divided into sectors stored as a sector link list (`fileptr`). Each entry in the sector link list contains the sector data (`data`) of 1024 bytes and a pointer to the next entry (`next`) in the sector link list.

Memory
Fragmentation

Users of the Oberon system usually read the complete file, for instance reading system modules, the compiler or the linker. The seek time for a specific sector of a file is shortened by maintaining information concerning the previous sector read for each file. On each file read the `hintsec` field is updated with the current sector number read and the `hintadr` field updated with the current sector pointer address in the file cache. When the requested sector number is equal to or larger than the `hintsec` field, the starting address of `hintadr` is assumed instead of searching from the initial address of the file in the cache. This prevents unnecessary searching through a possible large number of sector link list entries.

Efficient
Sector Seek

2.4 Directory Environment

To prevent frequent searches directed via CFS at the `TcpSVR` (to determine the existence of a file and its ownership), a directory cache is maintained. Besides this improvement in efficiency when attaining file existence, another key advantage gained from a directory cache is the improvement in efficiency when a user requests a directory listing of files.

Directory Cache Re-
quirement

In addition to the groups and system wide user directory entries read at service startup, directory entries of active users are also maintained in the directory cache. Unlike group users whose directory entries are removed only when cache space is required, a user's entries are removed from the directory cache as part of the logout request. (see `logincount` in Section 2.3). The directory cache is searched for file existence when a user opens a file. Once the existence and ownership of a file is determined, the user's file cache space is searched for the file entry. A `TcpSVR` request reading the file into the file cache is issued on file cache search failure. The file data is returned to the client if the file exists.

Efficient Directory
Cache Maintenance

The structure of the directory lookup table or directory cache, graphically represented in Figure 3, is similar to the cache lookup table (file cache). Each entry in the directory table represents a user (UID) and contains the following fields: `empty`, `day`, `hour`, `min` and `first`. The `empty` field ensures the existence of user files. A separate field is required because users who do not own any files might enforce an unnecessary number of directory listing requests to the `TcpSVR`. The time attributes (`day`, `hour` and `min`) indicate the last time of directory access and assists the cache replacement process discussed in Section 2.5. The `first` field points to the link list of directory entry sets.

Directory
Lookup Table

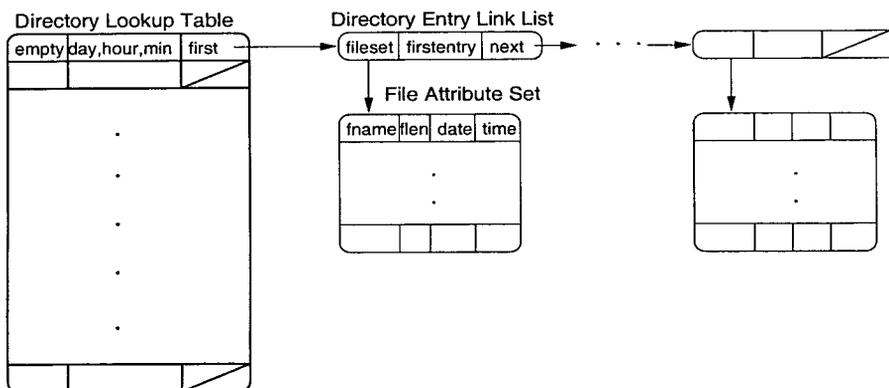


Figure 3: *Directory Entry Lookup Table*. Users directory entries are maintained, when logged on at CFS, in a Directory Lookup Table. These users have an entry in the table containing a Directory Entry Link List. Each entry in the link list contains a set of all file attributes (File Attribute Set) owned by the user. Groups and the system wide users directory entries are read into the directory cache at service startup and only removed when free cache space is required.

Each entry in the link list of directory entries contains a set of file attributes (**fileset**). The attributes are: **filename**, **filelen**, **date** and **time**. Included in an entry in the directory link list is a field containing the index (**firstentry**) of the first available entry in the set of file attributes. Finding the first available entry for a user requires a search through each directory link list entry until an available **firstentry** is found. A full set of file attributes is represented by a negative **firstentry** field. The **next** field points to the next set of file attributes.

To prevent memory fragmentation when frequent new files are created and memory for a file's attributes are allocated, a set of 93 file attributes is maintained. Calculating the size of each set of file attributes enables memory allocation on 4K page boundaries. This calculation includes the **next** pointer field.

2.5 Cache Replacement Policy

As the cache reaches full capacity, some mechanism must remove file entries to free cache space for new file entries. A cleanup thread in CFS is activated when cache space fills up. The thread uses a process similar to Least Recently Used (LRU) (see Section 3.2.3) as a replacement policy by, amongst other things, replacing the older file entries

in the cache with new file data.

The cleanup process executes in two modes, firstly in a non-volatile mode where file entries are removed from non-locked cache lookup table entries. Secondly, when necessary, file entries are removed from locked cache entries (volatile mode). The volatile mode ensures that cache space is made available when the server load is high or when malicious clients refuse to relinquish control of their respective cache table entries. All client threads are blocked while the cleanup procedure executes. Cleanup Policy

The cleanup process executes as a six step process, some steps executing in two different modes, executing each step until the requested memory is available. The outline of each step is shown below as part of procedure `SeekAndDestroy`. Some sections of code are described in high level terms using *emphasized* print. Comments are enclosed between “(*)” and “(*)”. The cleanup procedure requires the user identifier of the user that requested free space to ensure that the user does not occupy too much cache space. The amount of memory required is supplied as well as the mode in which the process should execute. The global state `EnoughMemoryAvailable` is reached when enough cache space is made available, ensuring that the cleanup process terminates immediately. Global Cleanup Procedure

```
PROCEDURE SeekAndDestroy (uid: INTEGER; memreq: LONGINT;
    volatile: BOOLEAN; VAR done: BOOLEAN);
VAR
    Time: INTEGER;          (* Indicate the age of files to remove *)
    cur_day,
    cur_hour,
    cur_min: INTEGER;      (* Current day, hour, min *)
BEGIN
    GetCurrentTime(cur_day, cur_hour, cur_min);
    Follow cleanup steps until state EnoughMemoryAvailable has been reached
END SeekAndDestroy;
```

The **first** step in the cleanup process removes all cache table entries of users who have not accessed the space within 24 hours. This ensures cache space for current users and is the only step which only executes in non-volatile mode. Steps two, four, five and six execute in both modes, while step three executes in volatile mode. Cleanup: Step 1

```

IF  $\neg$  volatile THEN
  IF DestroyDeadSessions(cur_day, cur_hour, cur_min, memreq) THEN
    EnoughMemoryAvailable
  END
END;

```

The **second** step is Oberon specific. The Oberon compiler creates a temporary symbol file each time a module is compiled. This step increases the available file cache space by removing the temporary symbol files before actual file entries are removed. The removal executes in both volatile and non-volatile mode because locked cache table entries may also contain temporary symbol files. Cleanup: Step 2

```

IF RemoveSymFiles(memreq, volatile) THEN
  EnoughMemoryAvailable
END;

```

The **third** cleanup step ensures the removal of stored file entries of the user that requested cache space. Entries are removed if the user occupies more than a quarter of the total cache space. This ensures some fairness to other users who may have just entered the cache space. This step is initially executed in a volatile mode because the user who requested free space will have a locked cache table entry. Consequently when the global volatile stage has been reached, this step is ignored. Cleanup: Step 3

```

IF RemoveUsersOldEntries(uid, memreq, volatile) THEN
  EnoughMemoryAvailable
END;

```

Directory entries require memory. Consequently the **fourth** step removes user directory entries when the directory table has not been accessed within a certain amount of time. Directory entries of all users, excluding group users and the system wide user are removed. The group users and system wide users directory entries contain important information required to boot the Oberon system. Therefore they are only removed after old file entries of users have been removed, as described below in step 6. Cleanup: Step 4

```

IF RemoveDirs(cur_day, cur_hour, cur_min, memreq, volatile) THEN

```

```

    EnoughMemoryAvailable
END;
```

The **fifth** step is the removal of old file entries to make way for new data. All entries that are removed have been updated at the storage server and are removed using a least recently used policy. Files older than a certain age are removed, decrementing the age as cache space is still required. Cleanup: Step 5

```

Time := 60;
WHILE Time >= 0 DO
    RemoveOldFiles(cur_day, cur_hour, cur_min, Time, memreq, volatile);
    IF EnoughMemoryAvailable THEN Time := -1
    ELSIF (Time <= 10) THEN DEC(Time, 1)
    ELSE DEC(Time, 5) END
END;
```

The **sixth** step is only reached when the pre-allocated cache space is small and the memory has become too fragmented or when few users fill up the cache space and refuse to relinquish cache space. This step removes the group users and system wide users directory entries and on failure to remove enough cache space switches the cleanup process into volatile mode. Cleanup: Step 6

```

IF ¬ EnoughMemoryAvailable THEN
    IF ¬ RemoveAllDirEntries(memreq, volatile) THEN
        (* Redo process in volatile stage *)
        ¬ EnoughMemoryAvailable
    END
END;
```

EnoughMemoryAvailable

2.6 Physical Disk Storage Service

Communication with the disk storage server occurs when CFS does an IPC request, supplying the necessary information, to the TCP thread in CFS. Because IPC requests TcpSVR Functionality

are buffered in the microkernel, single access to the `TcpSVR` is ensured at the `CFS`. Besides sending and receiving file data and information, the storage server also provides functionality to manage users (create and destroy) on disk. Table 5 in Appendix B shows the operations supplied by the disk storage server.

At startup time the `TcpSVR` ensures the validity of the file system directory on Linux `TcpSVR Startup` where the files are stored. Each user's files are stored in a separate directory within the file system directory. Once the validity of the file system directory has been determined, a TCP socket is created [Ste98]. On connection a child process is forked, a new socket created for the child and communication commences on the newly created socket. The parent process may receive more incoming connections from other `CFS` VMs. This ensures that extending the prototype file service to a multi-server environment would only require changes to `CFS`.

Several utilities were written to accompany the prototype file service. A remote console VM (`CFSConsole`) provides maintainability of `CFS` from any machine running in the distributed environment. Certain operations issued by `CFSConsole`, for instance removing a user's files from the cache, require the `SysAdm` password. An Oberon utility module (`CFS.Mod`) provides information concerning `CFS` from within an Oberon client. Other operations provided by this utility include renaming a user's current password and copying files between different `CFS` user accounts. `CFS Utilities`

In total `CFS` consists of about 5000 lines of Oberon code, while the `TcpSVR` consists of 950 lines of C code, excluding system libraries. The `CFS` utilities consists of about 1000 lines of Oberon code. The code image of `CFS` is about 220 K. The current `CFS` VM provides cache space of 250 MB, which is more than enough for the current user load. `Code Size`

The file service is currently in use and provides a suitable environment for accumulating performance data. Some changes, necessary for trace collection, were made. These changes resulted in the accumulation of sequences of file system requests, and where necessary, the relevant block addresses. Traces collected were used to test an improved server `BFS` described in the next chapter. `File Service Goal`

Chapter 3

A Block Storage Service

The file service (CFS) presented in the previous chapter has some disadvantages. Linux is required for physical storage which causes additional communication overhead and when new cache space is required, an expensive cleanup procedure reduces the usefulness of the service. This chapter presents an improved server (known as BFS) which addresses the inefficiencies of CFS. Three key issues were kept in mind during the design of BFS: efficiency, reliability and availability. Storage Service Requirement

The design philosophy of the block storage service is to separate the management of blocks from file management. Files are managed by the client machines while disk blocks are managed by a specialised server machine that is shared among many clients. One advantage gained from this separation is flexibility: different file systems can be supported while the block server manages only blocks. The block server was designed to be stateless to simplify recovery after a system failure. Some design issues concerning the storage service are presented in Section 3.1, while Section 3.2 presents efficiency issues concerning the service and describes different cache replacement strategies. Section 3.3 presents reliability and availability as design criteria. Some implementation issues are presented in Section 3.4. Design Philosophy

3.1 Design Overview

The block storage system is designed to provide an efficient, reliable and flexible service to clients. Traditionally information that describes the structure of files is maintained Service Overview

by the server. Client requests are addressed at files. Typical requests involve opening, closing or deleting as well as reading and writing files. By providing a block service instead of a complete file service, more flexibility is achieved. Multiple clients, using different types of file systems, can use the block service. The block storage server is designed to accept connections from the Oberon file system and the Linux file system.

3.1.1 Stateless Service

A necessary design consideration is that of *client reconnection* on service failure. As is the case with most reactive systems, they sometimes fail due to unforeseen conditions (e.g. power failures). In these situations clients connected to the system lose the connection and are required to reconnect to the system after failure, losing the data associated with the previous connection if not saved on a permanent medium.

Tanenbaum [TvR85] defines a file service as stateless when client supplied operations on files identify the file and the position of operation in the file. In the stateful approach the server maintains an identifier for each opened file. This identifier can be the filename or, at times, a shorter unique number generated by the server. Clients and the server use this identifier to distinguish between files. All operations performed by clients on the opened file provide the file identifier to the server for correct file identification. On service failure all such state information is lost by the server and clients are required to reconnect to the service and rebuild their file identifier information. For clients to rebuild this state information, each previously opened file for each client will require re-opening. This can be a tedious and time-consuming process, especially when many clients were connected to the service with many files opened before service failure.

A possible solution in providing a service for clients to reconnect to is *checkpointing*. The goal of checkpointing is to establish a recovery point in the execution of the program and to save enough state information to restore the program to the previous saved state. Checkpoints for distributed systems store global system state to physical storage. A log is maintained on physical storage of all state information that is required if the service fails. When the service is restarted after failure, this information is retrieved before clients can issue requests. Wang et al. [WHK97] provide progressive steps of rollback recovery that use the logged state information to regenerate the previous state of a program before a fault occurred. Although this recovery technique is not of importance here, checkpointing shows an increase in program execution time of up to 10%. Plank

et al. [PCBK99] reduce the program execution time with a method called *memory exclusion*. This is a state size reduction technique where state that has changed since the previous checkpoint operation is stored, ignoring unchanged state. This technique is enforced by ignoring memory locations which have not been read at checkpoint time. For a reactive system like a storage server checkpoints would require the storage of file identifiers and their respective associations (file owner, file name etc.) on each new file identifier allocated or deallocated. Although memory exclusion can be used to reduce the logged state size at each checkpoint, a performance penalty still occurs for each storage update.

Another popular solution to the problem is replicating data between various servers. On server failure another server is contacted to obtain the requested data. An example of such a service is the Coda file system [SKK⁺90] which replicates file data between servers. A list of available servers is maintained by each client and on each file update all servers are notified. This maximizes the probability that every replicated server has the current data. File data conflicts are resolved by the servers. An alternative approach to server replication is to connect a backup disk to another server. This is known as *dual-ported* disks [BEMS91]. The main server stores its volatile state information on a disk log. On server failure the backup reconstructs the server's state from the log and impersonates the main server. Unavailability is only experienced while the backup reconstructs the main server state. An immediate problem to the replicated data approach is that of version control. As file data is updated, different file versions at different locations (servers) may exist thus requiring frequent file updates at all locations. Experience with CFS concerning efficiency has shown no requirement for server replication, hence the problem of distributed block replication need not be solved. Bhide et al. [BEMS91] have shown that when writing a 4K file an overhead increase of 21% is achieved when maintaining log information concerning the write and mirroring the file to the backup server. This disadvantage was considered too serious to consider this technique.

Replicated Data

An alternative approach is proposed and motivates the design decision to make a clear distinction between files and blocks. Clients maintain files locally in their individual file systems thus maintaining critical state information like file identifiers. The storage service only identifies blocks. This stateless approach requires client file systems to supply the address of the block on which the operation is performed during normal service operation. On service failure clients reconnect to the failed service much faster

Stateless
Approach

by repeating the last failed operation. This is simpler when the server is designed to manage blocks instead of files. No critical information is lost by the server because state is maintained by the client file system. No additional performance penalty is paid as is the case with checkpointing and the replicated data approach.

The approach followed by making a clear distinction between client file systems and a block service allows greater flexibility towards different file systems. The smallest unit of allocation in a file is a block. By providing a small set of simple operations on blocks most file systems will be able to connect to the storage service with little change.

Flexibility

3.1.2 Blocks

Disks are divided into small allocation units called sectors. The unit of storage used by the server described here is a *block* of data which consists of a number of disk sectors. The structures defining the abstraction of files and their location on disks differ from file data blocks. The block storage service does not make a distinction between these structures defining files or *file system blocks* and the file data blocks and includes both as *data blocks*. Furthermore, the block storage service defines non-volatile storage disks as devices divided into data blocks and allocation blocks describing the allocation state of data blocks. Figure 4 shows a graphical representation of the different types of blocks used by the block service. A *logical block address* is the address of a data block on a storage device, thus excluding allocation state blocks and other blocks describing the block service. The logical block address space differs from the normal *physical block address* of the storage device which include all device blocks. The block size is important, because a small block size will require more disk seeks, while a large block size may waste disk space. [TW97]

Device Blocks

Storage services require provision for block allocation which is handled by a single entity that allocates blocks on behalf of all clients.

Allocation Problem

A possible solution in providing block allocation to clients is to maintain a list of free blocks [TW97]. The advantage gained from such a solution is that as disks fill with data, less storage space is required to indicate free space. However, for large disks which are relatively empty, this technique is impractical.

Free List Solution

Some storage systems, for instance the Amoeba Bullet file server [vRvST89], use a method of *contiguous allocation* of space. The advantage of this method is that free

Contiguous Allocation

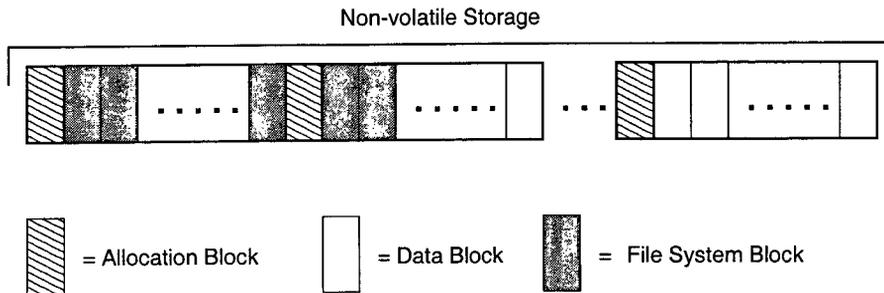


Figure 4: *Block Description*. The block service manages blocks on non-volatile storage devices. Devices consist of segments of allocation blocks describing the allocation state of data blocks. Data blocks can contain file data or the structures defining the data (file system blocks).

space can be stored as an address and an additional number of contiguous available blocks. The disadvantage is that as the disks fill up it becomes more difficult to find contiguous allocation space. Another disadvantage is that the storage server must know the number of blocks to be allocated contiguously. This is a difficult calculation. Even at file level calculating the size of an output file beforehand is a difficult estimation. Even if clients know the number of blocks all file data must be cached until the complete file can be written. This limits the number of open files a given client can have opened simultaneously.

The proposed solution is for the server to maintain the allocation state of every block on every non-volatile storage device, instead of maintaining all free blocks. Although maintaining the allocation state of every block on large disks sounds daunting it is still more practical than a free list. A single bit is enough to record whether a given block has been allocated or not. Storage space is saved compared to a free list implementation. Some efficiency issues concerning disk allocation are described in Section 3.2.2.

Proposed Solution

3.1.3 Partition Table

To provide the abstraction of a file, disk blocks are combined together. Some storage systems maintain a file in consecutive disk blocks, while other systems place each file block on disk at a random location. Files are defined in structures stored on non-volatile storage media. These structures and the file location on disks are maintained by a file system. The location and layout of the structures defining the position of each file on

Problem
Description

disk differ for different storage systems. For example, in UNIX each file is represented by an inode block [SG94]. The inode contains information such as the owner of the file and its allocation state, storing physical disk blocks in the structure. Large files are addressed via multiple inodes. Inode structures differ completely from other storage system structures such as File Allocation Tables (FAT) defined by the MS-DOS file system [Dei90]. FAT tables contain a linked list of block addresses belonging to each file. No file ownerships are defined; only file attributes indicating the type of file (such as system or read-only). Furthermore, various file system structures differ in size, in data layout and in disk layout. File systems also use different block sizes. A server designed to manage blocks as used by different file systems must be able to handle such differences effectively.

A reasonable approach is to store each user's files in its own segment of blocks irrespective of the file system used. The file system is stored separately in its own segment while user data blocks are stored in other segments. The disadvantage of this approach is that each user and each file system segment is limited to a pre-determined segment size. Another disadvantage is the inability to use certain user hierarchies. Recall that the user hierarchy of CFS enabled users to read certain user files but not all. To enforce this hierarchy on this design is too difficult.

User Approach

Different file systems should be stored on different abstract locations together with the file data blocks to prevent inconsistent data returned to clients. A possible solution is storing each file system and relevant file data on a separate storage disk. The disadvantage of this approach is the limitation it places on the number of file systems. For each disk only one file system is allowed. Another disadvantage of this approach is the enforcement of the disk capacity per file system.

Disk Approach

An alternative solution is proposed. Each disk is divided into smaller segments or *partitions*, each partition containing structural information and file data blocks. An immediate advantage is that the number of file systems are not limited to the number of disks but to the number of partitions. Although this limits the capacity in the same way, partitions can span more than one disk, providing more flexibility in the size of each partition and the number of partitions. Storage space for a specific file system and data is limited to the size of the partition, but can be extended when full capacity is reached by adding another partition. Adding another partition at runtime is also possible. This can be implemented by selecting a new partition size and locking

Partition Approach

incoming requests for that specific file system until the partition has been created.

Storage servers are usually designed to accommodate one or more disks. A file system is created on the disk allowing multiple clients to retrieve and store file information. If additional space is needed, more disks can be added to extend the file system. CFS (see Chapter 2) is an example of such a service. Our approach with the block storage service differs from the traditional where all disks associated with the block storage service combines, in conjunction with a block cache, into one *virtual block store*. This limits storage space to all disks and all available memory for the block cache. Four main data structures are needed: a cached partition table, a block cache, a block memory structure and a bitmap cache.

A partition table is maintained by the virtual block storage service on the first disk to address each partition and its storage device location. Each entry in the partition table contains a file system identifier, the partition number associated with the file system, the block size of the partition, the start address and the size, in blocks, of the partition as well as a device identifier associated with the storage device. Because different file systems may use different block sizes, the block size of the partition is stored in the partition table. Clients are allowed to access a partition by supplying a pre-determined file system identifier associated with a partition and logical block address. This information is validated against the partition table information maintained, in full, in a write-through partition table cache, the first main data structure of the virtual block storage service. A change in partition layout must result in an immediate change on disk. A service failure will cause a lost file system extension if the change is not updated on disk.

The remaining three data structures used by the block storage service are the block cache, a block memory manager used in conjunction with the block cache as well as a bitmap cache. All three structures are described below.

3.2 Efficiency

An important design requirement of a storage service is *performance transparency* [CDK94]:

Client programs should continue to perform satisfactory while the load on

the service varies within a specific range.

Storage server performance is improved by reducing the number of costly disk operations. Results obtained from CFS showed a read hit ratio of 94%.¹ Since it is well known that service performance will improve when caching data, a block cache, the **second** data structure, was designed and implemented. Although several different data structures could be used, one that can address many disk blocks and is superior in search speed was chosen: a hash table [Knu98, ED88].

3.2.1 Block Cache

The Sprite distributed file system [LS90] is an example of a system that was specifically designed for diskless client workstations with large main memories. The file system caches block data (4K) at servers and clients and addresses blocks in the cache virtually with the use of a file token obtained when the file was opened and a block offset from the initial token from the file. This prevents the necessity of acquiring an inode address from a server when accessing the client cache. No read-ahead scheme exists but a write-delayed approach is used on file modification. Blocks that have not been updated to physical storage (dirty blocks) are flushed to the server every 30 seconds or when the block is removed from cache space. When a client opens a file, a version number associated with that file is returned. The client compares this number with blocks in its cache belonging to the file. If the version numbers mismatch the blocks in the client cache are discarded and reloaded from the server when required. Because of the write-delay policy an explicit open on a file forces the server to contact the last writer to the file and enforce all dirty blocks contained in the last writer cache to be flushed to the server. Consequently servers maintain a list of all last writers to files. When the server detects that one client is writing to a file while another client is reading from it, client caching for both clients are disabled. Thus all requests concerning the two clients are channelled through the server.

Example 1:
Sprite

A second example of a storage service that makes extensive use of caching is SUN Microsystem's Network File System or NFS [CDK94]. NFS caches both at client and server side. The server cache maintains the standard UNIX buffer cache where a read-ahead scheme tries to anticipate future reads by inserting blocks in the cache following

Example 2:
SUN Network File
System

¹Results extracted from CFS showed that the number of read calls made by clients was 359657 with 338437 read hits logged. 91323 write calls were logged.

the most recently read blocks. No delayed-write scheme is used by NFS as is the case with the UNIX buffer cache. This write-through server cache ensures data consistency on service failure. The client cache maintains a timestamp for each file. On reading the file a validation check compares the timestamp with a service file modification time. If the modification time is more recent the client blocks are invalidated and the blocks are fetched when required. The validation check is performed when a file is opened or when a new block is read from the server. Writes use the same mechanism used by the UNIX buffer cache by marking the cache block as dirty and using a write-behind scheme by updating all the dirty blocks every 30 seconds with the UNIX *sync* call. When a file is closed all dirty blocks are flushed to disk.

To design an efficient block cache some issues require addressing. An important design issue is the method of storing and locating a requested block in the block cache. A possible approach is to maintain the logical block address in the cache. The disadvantage of this approach is that the partition identifier and the device identifier associated with each block are required to distinguish between blocks of different devices. A worthwhile solution is to store the physical block address in the block cache, thus saving cache space for block data. This is significant, especially for a large cache addressing many data blocks. Another important design issue of a hash table that influences service performance significantly is the size of the hash table. A table size too small to maintain all data blocks will result in too many collisions, resulting in an increase in search time to find the correct block. Consequently determining the correct number of link list entries in each hash entry becomes important. The search time through a complete link list is a linear function of the number of entries in the list. On the prototype implementation, searching through a link list of 50000 elements requires more or less the same time as an average seek operation: (11 milliseconds (ms)). An equally important design issue is concurrent client access. Because the service may maintain connections from several clients, provision should be made for concurrent access to the cache space. It is possible for clients to access the same hash entry area. A possible solution is for clients to retransmit their request. Unfortunately this will increase network traffic. A solution is to block a client's request at the server until the initial client's request in the block cache is finished. On completion, the initial client signals the release to the blocked client.

Cache Design

An important design issue of a cache is the prevention of frequent allocation and deallocation of memory for the block cache. This requires time and memory may become

Memory
Fragmentation

fragmented. Therefore less continuous memory space will be available. To prevent this, a block memory structure or *block memory manager*, the third main data structure of the virtual block storage service, used in conjunction with the block cache, stores block data without frequent allocation and deallocation. The memory structure is allocated at server startup time with the block cache. By maintaining an indexed array of blocks in memory, operations can use the index to find the required data. A memory bitmap is maintained (first-fit strategy) to define the index space. When the block memory (cache) fills up, the removed block index is used by the new block, instead of searching through the memory bitmap.

The block cache environment, graphically represented in Figure 5, stores block data in an open hash table. A simple calculation supplies the position in the hash table of a block entry (block data). To prevent concurrent access by several clients a locked parameter is set, preventing one client from accessing the hash entry until the initial client has finished. The physical block address is maintained in field `secno`. The block data is stored in the block memory structure, the index (`memadr`) calculated via a memory bitmap structure with a first-fit strategy.

Block Cache Layout

3.2.2 Bitmap Cache

This section describes efficiency issues concerning the allocation data stored on disk. To prohibit frequent disk access when reading or updating block data, the allocation state of disks is sometimes cached. UNIX is an example which uses an inode cache that maintains recently used inode blocks that describe files in memory [CDK94]. The inode block is cached until space is required for a new inode block. The cache uses read-ahead to obtain possible future references and write-delayed similar to NFS described previously in Section 3.2.1.

Example 1:
UNIX

Another example of a storage service that caches allocation state is the RHODOS distributed file facility. Each server in the RHODOS file facility maintains a disk server which uses bitmaps to describe the allocation state of disks. In addition to this a two dimensional array is maintained to quickly obtain a requested number of contiguous blocks. The first row in the array stores references to single free fragments. The second row references a group of two contiguous fragments. Similarly for row three and so on. On request for a specific number of contiguous fragments, a quick lookup is done in the two dimensional array.

Example 2:
RHODOS

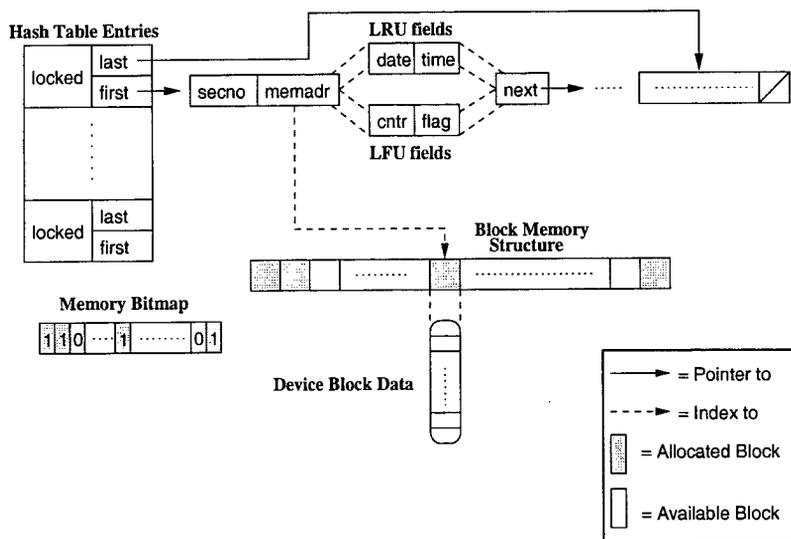


Figure 5: *Block Cache Layout*. The block cache hash table is implemented as an open hash table, each hash entry (**Hash Table Entries**) containing a link list of blocks. An index (**memadr**) in each link list entry provides an index to a pre-allocated memory structure. The pre-allocated memory structure is addressed by a memory bitmap. Allocation occurs on a first-fit strategy. The **LRU** and **LFU** fields are used by the respective cache replacement policy.

Recall that the virtual block storage service uses bits to describe the allocation state of all data blocks on disks. These bits are stored on each partition as bitmap blocks. To prohibit frequent bitmap block reads when determining the existence of a data block, the bitmap data of all partitions in the virtual block storage service is maintained in memory. The fourth main data structure of the block storage service is this table in memory known as the *bitmap cache*. A write-through bitmap cache, updating each cache change immediately on the correct partition, prohibits faulty service behaviour when the service is restarted after failure. If the service fails and the allocation state has not been updated on disk yet the data block can be overwritten when the next client requests an available block.

Bitmap Cache

If allocation data is located at the start of a large disk disk access time will be compromised. A write would require a disk seek to the required data block and to the start of the disk to update the allocation block. An alternative approach followed by some systems, such as UNIX [MJL84], is to distribute allocation data across disks to minimize disk seek time when updating a data block and the allocation block describing it.

Bitmap

Location Problem

In UNIX, disks are divided into cylinders and each cylinder group contains a bitmap describing the allocation state of that cylinder group. To prevent loss of data on disk failure the allocation state is distributed in the cylinder group.

A similar approach is followed by the virtual block storage service. Although disks are divided into partitions, the partitions can still be large. To compensate for this, bitmap blocks are distributed across each partition. An important efficiency issue concerning the bitmap data maintained on non-volatile storage is the size of the bitmap block. If the size of the bitmap block used is too large an unnecessary number of disk sector updates will occur when updating the bitmap block. The average file size measured on CFS was 11K when reading and 2K when writing. A bitmap block equal to one disk sector of 512 bytes can address 4096 data blocks. If a data block size of 1K (1024 bytes) is assumed, one bitmap block can address 4 MB of data blocks. Consequently a bitmap block size equal to one disk sector is more than sufficient for the Oberon file system to ensure that an average file write would result in the data block updates, but only one bitmap block (one disk sector) update.

Disk Bitmap Location

The virtual block storage service is designed to be efficient. Therefore client requests should get a fast response from the server. An equally important efficiency issue is searching through the bitmap cache addressing large partitions to find an unallocated data block. Slower response time will occur if the search time is slow. The search process may take some time especially if the bitmap cache is almost fully allocated. On the prototype implementation, a linear search through 300000 fully allocated sets of 32 bits takes 1497 ms. Tests on CFS have shown that up to 433 file system calls can be serviced per second. Considering this, a linear set search through a large number of sets is too inefficient. A binary search through each set in the bitmap requires 765 ms. A single test for an available bit in a set before performing the binary search lowers search time to 61 ms.²

Set Searching

The binary search process, used for searching through the bitmap cache and the memory bitmap, runs through each set in the search space until an available entry is found. The process does a binary search on each set (32 bits) by dividing the set in half after each recursive call. The process is performed until the left position in the set is equal to the right position, thus searching through the entire set. As mentioned before, a first-fit strategy is used, therefore each set is searched until an available entry is found. This

Search Description

²These measurements were obtained on a Pentium-100.

disadvantage of searching through the entire set is compensated for by making only one comparison after each recursive call.

When the cache fills up, some mechanism must replace another block. The *optimal replacement policy* would be to replace the block which will not be used for the longest period of time. Unfortunately this cannot be implemented for knowledge is required about future disk access. Two well-known replacement policies (see Chapter 9 in [Dei90]) were implemented for the hash table block cache and compared.

Cache Block
Replacement

3.2.3 Least Recently Used (LRU) vs. Least Frequently Used (LFU)

An important efficiency issue of a block cache is which policy to follow when replacing a block in a cache that has reached capacity. This section describes and compares two well-known policies: Least Recently Used (LRU) and Least Frequently Used (LFU). The methodology behind the LRU replacement strategy is that of *temporal locality*: blocks that have been referenced in the recent past will be referenced again in the near future. Consequently blocks in the cache are time-stamped and when necessary the oldest block is removed. The LFU block removal policy assumes that blocks with a high reference count are more likely to be referenced again than blocks with a low reference count. Consequently, each block should maintain a reference count. This frequency based policy does have some problems that require addressing.

Policy Methodology

A problem with the LFU policy is a sudden burst of references to one block in the cache, leading to a high frequency count. The result is that such a block would seldom be replaced even if it is never referenced again. Therefore a recently inserted, more important block may be removed instead of the block with the high frequency count. Some mechanism must prevent this. A solution to this problem is *aging*. A process ages the reference counter of each block in the hash table by dividing its counter by two, when an average frequency count has been reached. The efficiency of the LFU policy is influenced by the size of the average frequency count used during the aging process. When using a low average, files may age too quickly and vice versa. Consequently a large cache requires a large average frequency count. Section 4.2.1 in Chapter 4 evaluates different aging factors for different cache sizes. An equally important design issue concerning the LFU policy is when clients do a write on a block that is not in their local cache. A read request is issued first and after the block has been inserted in the client cache, a write commences. This will increment the block's reference counter

LFU Policy Issues

twice. Willick et al. [WEB93] provides a solution by not incrementing on writes and is enforced by using a flag parameter for each block in the cache.

Many implementations use the LRU policy. An example is the buffer cache of Minix which uses a doubly linked list of blocks sorted from most recently used (back) to least recently used (front). An open hash table chains different locations in the buffer cache together to optimize the search time for a required block. All blocks with addresses that hash to the same hash entry are chained together. When necessary, the the oldest entry in the hash table is discarded. A counter is checked to ensure that the block is neither in use nor a bitmap block which is not allowed to be removed. Blocks that will not be required soon such as double indirect blocks (double inodes) are inserted at the front of the list while other blocks are inserted at the back in true LRU fashion.

Example:
Minix

Caches maintained by servers differ from client caches according to the client reference stream of blocks. Server caches maintain data referenced by all clients, but references satisfied by client caches are removed as new blocks are requested. The reason for this is simple: recently referenced blocks will be in the client cache. Consequently blocks with a higher reference frequency are more likely to be referenced again than blocks that have been referenced recently. The LRU policy is known to be an effective policy for memory management [TW97, Dei90]. However, the least frequently used policy (LFU) is also claimed to work better for caches at client workstations in distributed systems. Trace driven simulations done by Willick et al. [WEB93] showed that the LFU policy outperforms LRU in general, except where a small client cache size is used. The client cache must be large enough to remove the temporal locality of the LRU policy.

Policy Comparison

An important issue that influences which policy to use is the amount of overhead it incurs. The LRU implementation searches through the cache space to compare the last referenced time of each block, discarding the oldest block. Instead of searching through the entire hash table, runtime efficiency is improved by maintaining a pointer for each hash entry to its last (oldest) link list entry. When a block in the cache is referenced, its position in the link list is shifted to the front and the block is time-stamped. A date field ensures that old blocks maintained in a large cache are removed at the correct time. When a block is removed all last pointers are compared to find the oldest entry in the entire cache space and after removal, the last pointer of that hash entry is updated.

Block Replacement
Runtime
LRU Hash Table

The LFU implementation searches through the entire cache space, comparing frequency counts of each block and discarding the block with the lowest frequency. Frequencies

LFU Hash Table

(counters) are updated as cache block are referenced or new blocks inserted. A write hit does not increment the cache block counter. A read miss or write miss initializes the block counter and, depending on read or write, the block's flag parameter is set. This flag parameter ensures that a read miss or write miss, followed by a read hit, will correctly increment the cache block counter only once. Although the age average is influenced when a new block is inserted (see state ReadMiss and WriteMiss), only a read hit, when a block's counter is incremented, enforces the aging process. The algorithm, executing in four stages WriteMiss, WriteHit, ReadMiss and ReadHit, is shown below as procedure UpdateBlockEntry.

```

PROCEDURE UpdateBlockEntry (blockstate: SHORTINT;
  VAR blockentry: BlockType);
BEGIN
  CASE blockstate OF
    WriteMiss:
      Increment the Age Counter;
      blockentry.cntnr := 1; blockentry.flag := TRUE
    |WriteHit: (* Do Nothing *)
    |ReadMiss:
      Increment the Age Counter;
      blockentry.cntnr := 1; blockentry.flag := FALSE
    |ReadHit:
      IF blockentry.flag = FALSE THEN
        Increment the Block Counter;
        Increment the Age Counter;
        Compute the Age Average;
        IF CurrentAgeAvg  $\geq$  ConstantAgeFactor THEN
          Age Each Block (* Divide all block counters by 2 *)
        END
      ELSE
        (* Ensures single increment after state WriteMiss *)
        blockentry.cntnr := 1; blockentry.flag := FALSE
      END
  END
END
END UpdateBlockEntry;

```

3.3 Reliability and Availability

This section describes more requirements of a storage service: reliability and availability. Design Criteria
The two requirements should not be confused with each other. *Reliability* is ensured by not corrupting or losing client data. *Availability* has to do with the service being there on request. Some design issues concerning both requirements are discussed in this section.

3.3.1 Reliable Service with Cache Coherency

When the content of the cache memory is identical to the data contained on physical storage or when the data is under tight enough control that physical storage and cache data are not confused with each other: the cache is said to be *coherent* or consistent [Han98]. Cache coherency is an integral part of a reliable system. An inconsistent cache may cause undesired results on service failure, for instance a client may be under the impression that a block has been updated on non-volatile storage. If the service failed before the block has been written to disk, then an inconsistency will arise. Coherency Problem

Many different caching techniques have been used and implemented in storage service software. A well-known example is the read-ahead and write-behind scheme used by most UNIX-like file systems [TW97, MJL84]. When processing a read operation, additional data is transferred from disk into the cache for future consecutive block provision. On writes data blocks in the cache are updated and marked dirty without updating physical storage. A process synchronizes the cache and physical storage on a regular basis by flushing all dirty blocks in the cache to disk. The problem with this approach is that inconsistent data between the cache and disk are lost during a system failure. Therefore a serious degradation in reliability occurs. In network file systems such as NFS [CDK94], server data is updated to disk immediately. Some systems use write-behind client caches placing the responsibility on the client to ensure that data is correctly updated. The UNIX Way

Reliability was considered important for the server described here. Consequently cache memory ensures fast reads but on writes most caches are write-through. This includes Our Approach

the client cache concerning file system updates (excluding normal data block updates), the block cache and the bitmap cache. This immediate update to physical storage ensures a high degree of service reliability, especially when the service fails.

The same degree of coherency is used for disk scheduling algorithms [TW97, SO90, TP72]. These algorithms reorganize blocks to be written for disk access time optimization. To use these algorithms in the virtual block storage service, clients must be willing to wait until the block has been written to physical storage. If the client is not willing to wait and the server crashes before the block has been updated, the client may be under the impression that the data has been correctly written to the disk.

Disk Scheduling Algorithms

3.3.2 Reliability with Stable Storage

Lampson [Lam83] defines stable storage as the process of eliminating all errors associated with a disk read or write. This is done by adding another physical storage medium to store a second copy of the original medium, also called *mirroring*. The price paid for this is that each block requires two write operations. The probability that the two storage devices might fail simultaneously is very low, but it still possible. Although this mechanism of stable storage is not failsafe, it eliminates almost all errors associated with physical storage.

To obtain the property of run-to-completion, Lampson [Lam83] defined *careful storage* which is implemented by an additional layer of software on top of normal disk operations. A *soft read* error means that an intermittent error occurs on the device. This usually occurs when the device is about to fail. *Careful reads* eliminate some of these soft read errors by repeating an operation until successful. If an operation remains unsuccessful after a number of attempts, it becomes a hard error. A *careful write* reads the block after it has been written and compares the two copies (read and written) for equality. If the two blocks do not agree, a *bad write* has occurred and the block is marked as “bad”.

Levels of Abstraction

Yet another layer of abstraction is built on top of careful storage to implement *stable storage*. The basic idea is to improve reliability by mirroring data onto a second storage medium. A *stable read* does a careful read on the original medium and on failure a careful read from the mirrored medium. A *stable write* does a careful write on both mediums in succession. On failure of the first careful write a server crash is enforced.

After every server crash (or after a preset time interval has expired) a cleanup process runs to ensure consistency between the two physical devices, comparing each block from the physical medium and the mirrored medium and enforcing block consistency.

Stable storage is not new. The RHODOS distributed file facility, for example, provides a distributed file service with some degree of file replication [PG94]. In addition to writing data on physical storage, clients of the RHODOS file service can specify whether data is to be replicated on a stable medium as well. Therefore the reliability of the service depends on the clients. An additional option provided by the file service enables a return from a file system write call before the second replicated update occurs. In addition to this, clients have the option of reading file data from the main or stable medium.

Example 1:
RHODOS

A stable storage implementation for UNIX [Any85] builds another layer of abstraction on top of stable storage. The added functionality of a stable object (file) is provided, updating the object on a stable medium as one atomic transaction in contrast to one atomic block update. This is done by means of a *moving tag*. After every successful update a tag, transparent to the client and calculated by the service, is appended to the file on a separate device block. This tag is recalculated and appended on each file write, hence the use of the word moving. An illegal file update will have no tag, consequently a crash recovery routine will be able to restore the consistency of the stable object. The recovery routine runs through all files to ensure consistency. If an inconsistent file is found, in other words a file without a moving tag, it is replaced with the consistent copy. Initially only tags are read and compared but after successful comparison files are also read and compared. The drawback of tags is that for each object an additional disk block is required. This wastes storage space.

Example 2:
UNIX

Another example where reliability is enforced is in the Amoeba distributed operating system [vRvST89]. The Bullet File Server, designed specifically for high performance, is an immutable file store that caches files. Therefore clients read and write complete files. On file update a new version of the file is created and sent to the server. Client can specify whether the file system call must return immediately, when the physical update is complete or after the file is replicated to a second file backup device. Therefore reliable mechanisms are provided by the server but the client is responsible for enforcing reliability.

Example 3:
Amoeba

3.3.3 Storage Cleanup

Recall that a cleanup process validates two physical devices for consistency. This process is required to ensure that a situation where disk data is inconsistent with the mirrored medium is fixed at an early stage. The process, similar to the UNIX file recovery mechanism described in Section 3.3.2, validates each partition on each storage medium to ensure consistency. Because the virtual block storage service is unfamiliar with files consistency is enforced at block level. This section describes the standard cleanup process used by most block stable storage implementations and some extensions specific for the virtual block storage approach.

Cleanup Methodology

The standard cleanup process used for stable storage performs a careful get on both the normal block and the data block on the stable storage medium. An error when reading a block is fixed by overwriting the data with the other block's data. If the two blocks are unequal, one block is overwritten with the other block. Both blocks cannot become bad simultaneously. If the normal block decayed, the cleanup process will have fixed the error in time before the stable storage block has decayed. This decay avoidance is enforced by executing the cleanup process when the server is not busy. The main section of the cleanup code is shown below.

Cleanup Section

```
CarefulStorageGet (StableStorageBlock, SSdata);
CarefulStorageGet (NormalStorageBlock, NSdata);
IF ErrorReadingStableStorageBlock THEN
    CarefulStoragePut (StableStorageBlock, NSdata)
ELSIF ErrorReadingNormalBlock THEN
    CarefulStoragePut (NormalStorageBlock, SSdata)
ELSE
    IF ErrorComparing(NSdata, SSdata) THEN
        (* Replace Normal Storage Block *)
        CarefulStoragePut (NormalStorageBlock, SSdata)
    END
END
```

The cleanup procedure is extended to provide the ability of copying bitmap block data on service startup to the bitmap cache. This ensures that bitmap data blocks are only read once. A `dataidx` variable, initialized to zero, is used as index to the

Cleanup Algorithm Extension

memory location where the data is copied to. After each copy instruction, the index is incremented by the copied number of sets representing a bitmap block. The outline of the code is shown below.

```

IF CopyBlockToBitmapStructure THEN
  IF ErrorReadingStableStorageBlock THEN
    CopyBuffer (NSdata, BitMapData[dataidx], SectorLen)
  ELSE
    CopyBuffer (SSdata, BitMapData[dataidx], SectorLen)
  END;
  INC (dataidx, SectorLen DIV (32 DIV 8))
END

```

A possible improvement in efficiency when performing a careful write operation is to modify the disk device driver to execute both the block write and block read, combined with the block comparison as one atomic transaction. This might prevent another read or write operation from moving the disk heads between the write and the read of a careful write causing additional disk seek for the read.

Careful Write

3.3.4 Service Availability

This section describes some issues concerning the availability of the storage service. Approaches to increase service availability are given which include an improvement to the standard cleanup mechanism. The improvement ensures a higher degree of availability for the virtual block storage service.

Introduction

A feasible approach to provide a service with a high degree of availability is server replication. An example of such a service is the Coda file system [SKK⁺90, CDK94]. This file system aims to provide clients with a shared file repository but to allow them to rely on local resources in case the repository is unavailable. Therefore file data is replicated between various servers whereby a list of available servers is maintained by each client and on each file update all servers are notified of the update. This maximizes the probability that every replicated server has the current file data. File data conflicts are resolved by the servers. Although server replication might provide a higher degree of service availability, no such requirement is necessary if the server restarts quickly

Server
Replication

and clients can reconnect to it after failure. An advantage gained when using a single server approach is that the problem of distributed block or file replication need not be solved. Only one authority maintains concurrent access to file data.

In single storage server implementations like the virtual block storage server, crashes degrade the availability of the service significantly. On server failure one should ensure that it becomes available as quickly as possible. The stateless nature of the service will ensure that when available, clients will all be able to reconnect. The most time-consuming process during service startup occurs when the cleanup process executes. Tests were performed on two small disks to obtain the time the cleanup procedure took to complete. The cleanup procedure required 13 minutes, lowering the availability of the service substantially.

One solution to the problem is to provide an interruptible cleanup process, allowing clients to read or write blocks to the service. This solution will require the relevant block to be checked for consistency in the normal cleanup fashion before the block is written or read. When multiple client requests occur while the cleanup process performs, the process will 'thrash' under possible heavy user load requiring a considerable amount of time to complete. Although this is the case at service startup, the interruptible solution is still valid when the cleanup process executes after a certain time interval when the service is not busy.

To improve availability on service startup the cleanup process does not execute. Instead the standard model of stable storage is improved by relocating blocks when a bad read or write occurs. A bad stable read from the original physical medium results in relocating the block from the mirrored medium to a new block on the original medium as well as the mirrored medium. The old block is already marked as in use and a scheme similar to that used by Linux uses is implemented to store the bad block address somewhere pre-allocated on disk. A bad stable write results in the allocation of a new block with the old one marked bad on both mediums. Any block relocation may require a file system change. Consequently the client file system must monitor any changes made to the logical block address sent, and update its file system structures. This change would result in the correct update at the storage service. Consequently the expensive cleanup mechanism is not required at startup. Blocks may still deteriorate simultaneously if they are not referenced frequently. To prevent this moderate deterioration, the cleanup process should still execute frequently, only when service load is low.

Problem
Description

Interruptible
Process

Improving Availabil-
ity

In addition to this the cleanup process is still used at service startup to validate consistency of the bitmap blocks. This validation is not interruptible because the server is then required not only to maintain client requests but to ensure block existence. This might require another disk access if the bitmap block has not been read from disk into the bitmap cache.

Service Startup

3.4 Service Functionality

This section describes various design and some implementation issues concerning the virtual block storage service (BFS). The virtual block storage service provides a simple but effective interface via a block cache to non-volatile storage (shown in Figure 6). An important design issue addressed by the service is reliability. Therefore the time between updating the data block and the file system block is minimized. The approach taken here is that write-through client caches are used when file system blocks are updated and on data block updates the cache may be write-behind. This ensures consistency between client file systems and the block storage service. Client requests are serviced by the storage server via transaction handler threads. To accommodate time-consuming operations on non-volatile storage devices, more than one thread is created to service requests. Besides managing concurrent access by clients, the block storage server maintains a block cache and partitions (including the bitmap cache) in the main thread. Although security is an important issue, the focus of this experiment is: efficiency, reliability and availability. Consequently security issues fall beyond the scope of this discussion.

Service Description

Before clients can contact a storage service, the service must be initialized. Firstly the functionality of the non-volatile storage associated with the service is ensured. At service startup, BFS detects the non-volatile devices associated with the block service and on success initializes the block cache data structure. The storage service must know the physical layout of the non-volatile storage, consequently the partition table is read into memory. To service client requests faster, the partition allocation state is copied from non-volatile storage into the bitmap cache. To ensure the correct functionality of the service while in use, a small console interface that displays errors and prints debug information is created in a separate thread before normal service operation is started. In the distributed environment clients contact the NameSVR to obtain a server

Service Startup

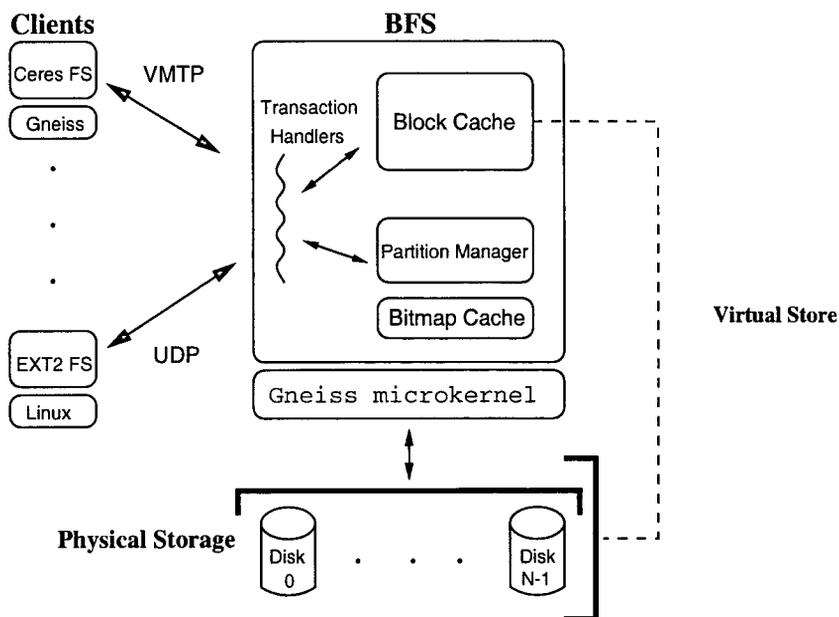


Figure 6: *Virtual Block Storage Service Environment*. The block service manages blocks in a block cache and when necessary requests management for the bitmap cache. Non-volatile storage devices (Disk 0...Disk N-1) are connected to BFS. The block cache, in conjunction with physical storage, form a virtual storage device. Clients connect from different machines via VMTP or UDP.

address. After all service data structures have been initialized, the storage service registers its address at the NameSVR. Transaction handler threads are created by the service to receive incoming requests from clients.

Clients communicate with a storage service by providing the necessary information to complete an operation. The virtual block storage service receives the client supplied request in a structure called `HeaderType`, graphically represented in Figure 7. In conjunction with this structure, an optional data block is supplied for writes. To make provision for server and client compatibility a `Version` field is included. As updates are made to the service, this parameter is set accordingly. Errors reported by the service are inserted in the `Error` field. As described in Section 3.1.3 clients provide a file system identifier that the service requires to compute the partition on which the specified logical block address is.

Client Header

As mentioned before the virtual block storage service is designed to be accommodate different types of file systems. Due to this and the stateless nature required of the

Flexible Operations

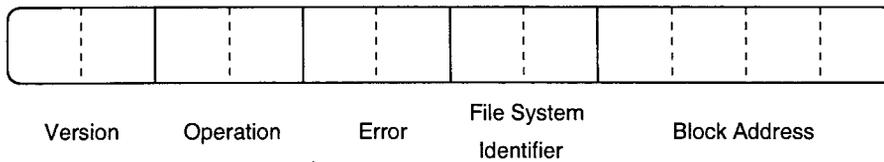


Figure 7: *Virtual Block Storage Service Header Type*. The block service (BFS) receives client requests in a structure called `HeaderType`. This type contains fields to indicate server and client compatibility (`Version`), the `Operation` to be performed, the `Error` that occurred, if any, the file system the operation is to be performed on (`File system identifier`) and the `Block address` in the file system.

service only three operation primitives are defined: `ReadBlock`, `WriteBlock` or `DeleteBlock`. Clients supply the operations in the header. The three operation primitives are described below. Operation descriptions are printed in *emphasized print*.

The `ReadBlock` operation determines the device physical block address by adding the client supplied logical block address to the specified file systems partition starting address contained in the partition table, taking the allocation block addresses of that partition into account. When a cache read miss occurs, the block is read from physical storage and inserted in the cache. The error field is set after the operation is executed.

Read A Block

```
PROCEDURE ReadBlock (VAR header: HeaderType; VAR buffer: BlockType);
BEGIN
    Determine physical disk address from logical address header.logicaladdress;
    Read block from cache via bfsCache module;
    IF  $\neg$  BlockInCache THEN
        Read block from physical storage via bfsPartition module;
        Insert read block into cache via bfsCache module
    END
END ReadBlock;
```

The `WriteBlock` operation writes the client supplied block to the physical block address on non-volatile storage and in the block cache. On device block update, the physical block address is supplied by the client, initially calculated by the client file system when the file system block was read from the block service. The existence of a block is validated via the bitmap cache. The logical block address is used for this validation,

Write A Block

calculated from the client supplied physical block address, because the logical address of a partition has a “one to one” mapping onto the bitmap cache. On creating a new block on a device, a physical block address is not supplied by the client and a new physical block address is calculated via the bitmap cache by the block service. The calculation includes searching through the partition’s bitmap cache to find the first available block for allocation (see Section 3.1.2). The new physical block address is returned, after block update, to the client. The client can then update its file system structures and enforce the file system update, using the write-through cache for file system updates, on the block storage service. A successful physical storage write ensures that data is written to the block cache.

```
PROCEDURE WriteBlock (VAR header: HeaderType; buffer: BlockType);
BEGIN
    Check validity of physical address via bfsPartition module;
    Allocate new or update old block on physical storage via bfsPartition module;
    Allocate or update block in cache via bfsCache module
END WriteBlock;
```

The DeleteBlock operation removes a physical block address, determined in the same way as the ReadBlock operation, from physical storage by updating the bitmap cache and the bitmap data structures on disk. On success the block is removed from the block cache. The removal of a data block will be followed by the file system update from the client.

Delete A Block

```
PROCEDURE DeleteBlock (VAR header: HeaderType);
BEGIN
    Determine physical disk address from logical address header.logicaladdress;
    Delete block from physical storage;
    Delete block from cache
END DeleteBlock;
```

The virtual block storage service is designed for use by several clients. Consequently some mechanism is required to prohibit concurrent client access to the same data. The user hierarchy used by CFS enabled user level locking as sufficient for concurrent access. Recall that in CFS a user’s file area is locked when his files are accessed. Alternative

Concurrent Access

solutions are required for the virtual block storage service because the service manages blocks, not files. File locking [PG94] or file leases [Gra90] have proved to be sufficient solutions to the problem. A file lock prohibits clients from accessing a file when a lock is set on the file. A lock is set when reading or writing occurs on the file. Leasing occurs when a server provides a lease for a file which expires after a certain amount of time. An extension to a lease may be requested. Because the storage service is not familiar with files, locks and leases must be stored on disk via the client file system. The locks and leases can be stored as files, maintained by the client file system as data files are opened, written, read and closed.

To compensate for slow service from non-volatile storage devices for which another client must wait, several transaction handler threads are created. The threads receive client requests and service them according to the operation field supplied in the header. The following example is a transaction handler thread that accepts requests from Oberon file system clients.

Transaction
Threads

```
PROCEDURE TransactionHandler;
BEGIN
  Setup message descriptors header and buffer;
  IPC.ReceiveRequest(StorageServerPort, header, buffer);
  CASE header.operation OF
    ReadBlock: ReadBlock(header, buffer)
  |WriteBlock: WriteBlock(header, buffer)
  |DeleteBlock: DeleteBlock(header)
  ELSE
    Return with Illegal Operation error code header.errorcode;
  END;
  IPC.SendReply(header, buffer)
END TransactionHandler;
```

Due to the flexible design of the virtual block storage service, different client file systems may connect to it. Although a Linux file system was not connected to the storage service, the only foreseen problem would be which communication protocol to use. Linux client file systems will require an operation on the block level, but instead of operating on local non-volatile storage, a network request is sent to the storage service.

Example File
System

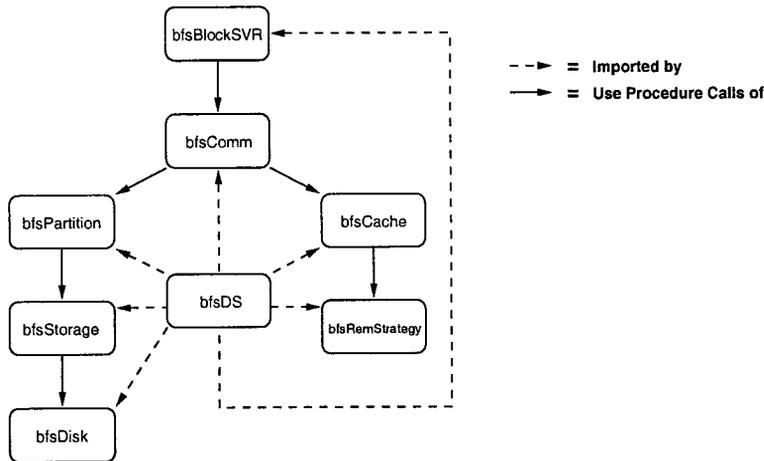


Figure 8: *Block Storage Service Module Layout*. The block service contains eight modules, written in Oberon. The `bfsBlockSVR` module initiates the service, while the `bfsComm` module enables communication. The `bfsCache` module provides a cache interface, with the `bfsRemStrategy` module providing several cache replacement policies. The `bfsPartition`, `bfsStorage` and `bfsDisk` modules provides separate interfaces for disk access. A `bfsDS` module provides all necessary type declarations and service constants.

Recall that in the distributed environment VMTP is used as communication protocol for the Oberon file system. A different communication protocol would be required for file systems which function outside the Oberon environment. The protocol chosen must be one supported by both the virtual block storage service and the underlying system of the client file system. Consequently Linux file system clients should communicate via TCP or UDP. A possible approach is to use TCP as communication protocol. To use this reliable communication protocol, a thread for each socket connection must be created by the virtual block storage service. If clients fail, the service thread created for the client remains in memory. Therefore too many connections may cause an unnecessary number of threads active in memory. Moreover, to destroy broken connection threads will be a difficult and time-consuming process. On comparing TCP with UDP valuable time is lost when socket connections are created by TCP between client and server. An alternative approach is to use UDP. Although UDP is an unreliable protocol, a more reliable service on top of UDP can be provided via timeout and retransmission algorithms ([Jac88, KP87]).

The virtual block storage service is implemented as 8 Oberon modules (see Figure 8) and consists of some 2000 lines of code. The VM size is 110K. Some development Code Size

utilities were implemented including a virtual storage manager used to create a virtual storage device. The utilities consists of some 600 lines of Oberon code. Figure 8 shows the Oberon module layout of the block storage service.

3.5 Summary

The virtual block storage server (BFS) functions in the same environment (see Section 2.1) as the caching file service (CFS), but differs in some ways. File Service vs. Storage Service

A distinction is made between files and blocks. Clients maintain files while the storage server manages blocks (1 or more disk sectors). On using blocks, a true stateless service is provided. CFS requires the maintenance of state information. Consequently clients need to re-login on service failure, losing unsaved file data. Only three operation primitives are required by BFS: Read, Write and Delete, enabling clients to reconnect on service failure. In addition, in some circumstances clients may not require a complete file. Therefore it is unnecessary to cache this file. Furthermore greater flexibility is achieved when using blocks instead of files. Different file systems can connect to the service. Moreover caching blocks is simpler than caching files. An inexpensive cache replacement mechanism is required to remove one block, while CFS requires an expensive algorithm to remove files. Blocks vs Files

In BFS non-volatile storage devices are connected to the server machine, while Linux is used as storage medium by CFS. Although using Linux does not require any block management, additional network overhead occurs between CFS and the TcpSVR. BFS provides a more efficient solution by not requiring any additional network overhead. Network Overhead

In BFS memory is initialized at service startup and memory references are managed with an index allocated via a memory bitmap. This saves time spent on allocation and deallocation of memory and prevents memory fragmentation, as is the case in CFS. Memory Management

The virtual block storage service provides a reliable service, mirroring each block on a similar physical medium. Some mechanisms are provided to maintain consistency between storage disks in the virtual store. These mechanisms perform a cleanup process at regular intervals by comparing each block with a mirrored block to ensure consistency. Although BFS provides a higher degree of reliability some performance decrease is expected due to the additional write for each block update and the confirm read. Reliable Service

CFS provides no reliability when a storage device fails.

BFS provides a high degree of availability. On service failure the server machine is rebooted and the service restarts as quickly as possible accepting immediate reconnections from clients. Improved mechanisms for the cleanup process include an interrupt feature to service unexpected client requests while performing the cleanup process, relocating bad blocks and booting the service without performing the expensive cleanup process before client requests can be serviced. Improved Availability

Performance measurements extracted from the virtual block storage service and compared with CFS are presented in the next chapter. Performance

Chapter 4

Performance Comparison

The approach taken in this study was to gather realistic trace data from a prototype file service and to use the gathered information to develop a virtual disk block storage service. Trace collection from the file service is described in Section 4.1. Section 4.2 presents a miss ratio analysis of reads, for both the prototype file service (CFS) and the virtual block storage service (BFS), using numerous cache sizes. Section 4.3 presents overall service performance with bandwidth and service response measurements of both the services.

Chapter Overview

4.1 Trace Collection

Realistic trace data were collected from the prototype file service (CFS) with the specific purpose of guiding it and the block service at different cache sizes. Changes were made to CFS which resulted in a performance difference, therefore the necessity to re-guide it. To guide the virtual block storage service (BFS) accurately, block information was required. Consequently changes were made to CFS to separately log block traces when guided with the initial traces.

File Service Modification

A test driver for CFS was implemented to gather file trace data. These trace data were input to a copy of CFS executing on a test machine. Because CFS caches files, each trace collected included the filename, the logical sector address of the file, the user identification number and the delay (ms) since the previous file system call.

CFS Trace Driver

A block driver gathers the relevant block traces from CFS which are then input to BFS.

Block Trace Driver

Because CFS only understands logical file addresses it gathers the physical data and inode block addresses from its storage server (TcpSVR) on both file reads and writes. Inode information is required to log file system block changes. The physical block addresses are computed by the TcpSVR using the GNU `libext2fs` library [Tso98].

The server machine used for both CFS and BFS was a Pentium-166 with 64 Megabyte (MB) RAM, using a 8003 Western Digital network adapter on a 10-Mbit Ethernet network. Two local disks, both Western Digital, were connected to the block service. The first disk (WD 23200) is 3.2 Gigabyte in size with an average seek time of 11 ms and a 256K disk buffer. The second disk (WD 1425) used as a mirror disk for stable storage (see Section 3.3), is 406 MB in size. A Pentium-100 with 32 MB RAM and a 8003 Western digital network adapter was used as the client machine. All tests were averaged through 3 test runs to compensate for random fluctuations due to other activity on the test machines and on the network.

Hardware Used

The initial traces were collected when six users worked on the file service generating 35300 reads in 31 minutes. This trace data was used throughout the performance comparison, except where otherwise stated.

Traces Used

4.2 Miss Ratio Analysis

Miss ratio analysis shows the most effective cache replacement policy when a cache reaches full capacity. In addition, it assists in indicating whether it is better to cache blocks or files. Section 4.2.1 does a miss ratio comparison of the BFS cache at different sizes for the least frequently used (LFU) cache replacement policy, executed with different age factors. To show the most effective cache replacement policy for our environment, Section 4.2.2 does a miss ratio analysis of the two replacement policies used by BFS, and the one used by CFS.

Overview

4.2.1 Comparison of Different Age Factors

In the previous chapter the necessity for an aging process, when using the LFU cache replacement policy, was identified (see Section 3.2.3). Recall that this process ages each block in the cache when an average age factor is reached. As the cache size is increased, more blocks can be inserted. Consequently different age factors may provide

Aging Recalled

better results for different cache sizes.

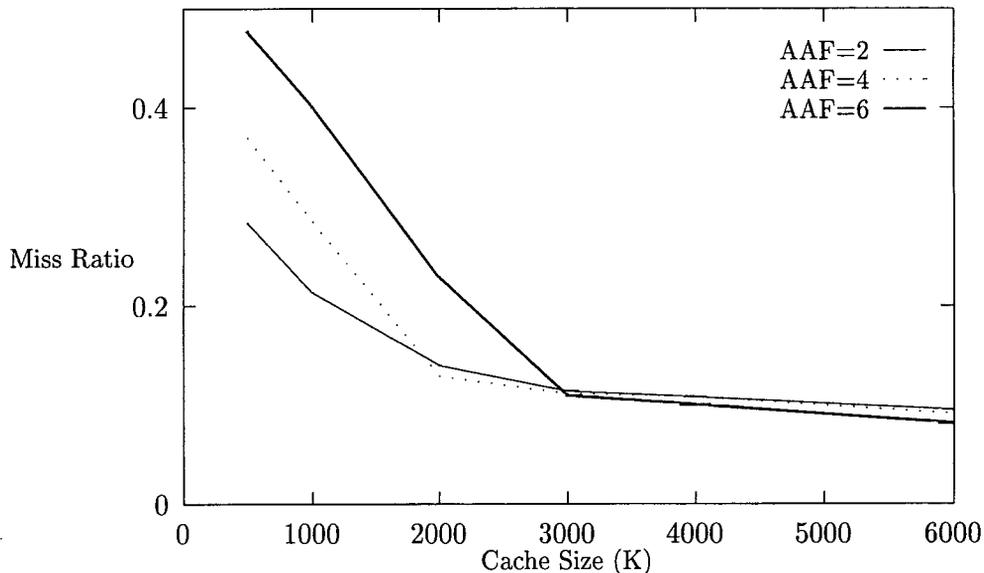


Figure 9: *Read Miss Ratio Analysis for different LFU Age Factors.* This graph shows the miss ratio analysis for different age factors when using the Least Frequently Used (LFU) cache replacement policy. An Average Age Factor (AAF) of 2, 4 and 6 is used.

Figure 9 shows the performance of the LFU policy for different average age factors (AAF). As the cache size is increased, the requirement for a bigger AAF becomes more apparent. All three AAFs converge at a cache size of 3 MB. The graph indicates that for caches below 2 MB a AAF of 2 is sufficient. For caches between 2 MB and 3 MB an AAF of 4 is ideal. An AAF of 6 is best for cache sizes of 4 MB and above. A good method for determining an AAF for a given cache size is dividing the number of blocks in the cache by 1000 for cache sizes above 3 MB. For caches below 3 MB use an AAF of 2.

Graph Observations

4.2.2 Comparison of CFS and BFS

Figure 10 compares the read miss ratio of CFS with the two cache replacement policies of BFS (LRU and LFU). The figure shows that CFS has a high read miss count when the cache size is small, but as the cache size is increased, the miss count decreases at a higher rate than that of BFS. The reason for this is simple: a small CFS cache prohibits the maintenance of many files in the cache and many cache misses occur because files

Graph Observations

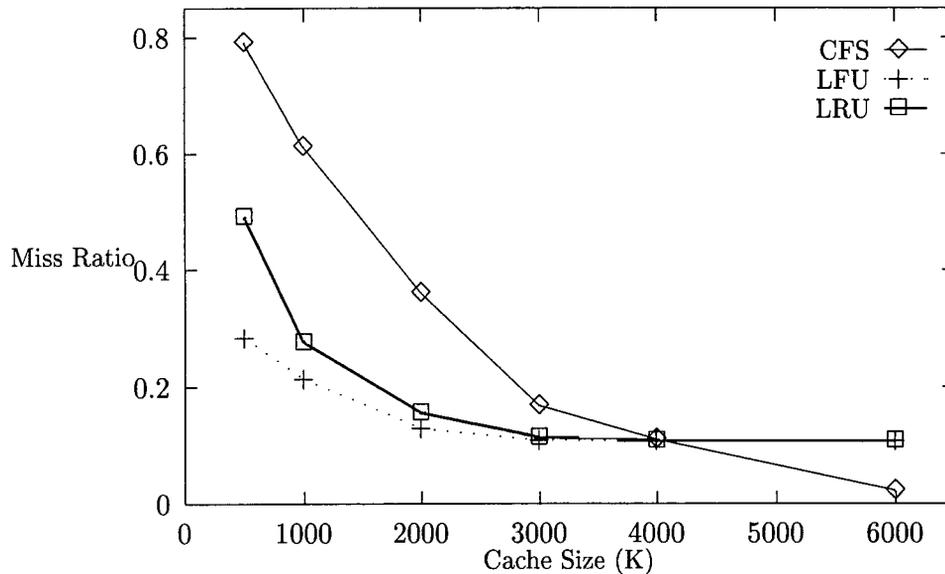


Figure 10: *Read Miss Ratio Analysis of CFS and BFS*. This graph shows the overall read miss ratio for CFS and for the two replacement policies, LRU and LFU, of BFS.

are frequently re-inserted into the cache. When the cache size is increased, more files can be maintained and more read hits occur, especially when the complete file is read by several users. Observing BFS, each block read registers a miss at cache insertion time, while with CFS only the first reference to a file registers a read miss. The minimum cache size for CFS under this user load is 4 MB, while 3 MB is sufficient for BFS.

A surprising result, concerning BFS, is that the LFU replacement policy is superior to LRU when the cache size is small, but as the size is increased the two policies converge. The runtime of each policy during block removal does differ significantly though. Although the LRU policy requires additional time when each referenced block entry in the cache is moved to the front of the list, the increase in time does not effect the overall runtime significantly. Measurements have shown that the average runtime of the LRU policy is 0.3735 ms, while the LFU policy registers a runtime measurement of 5.45 ms.¹ The difference in time indicates the runtime advantage gained by using a last pointer to compare each last hash entry instead of running through the complete hash table. This also indicates that it is better to use the LFU policy for a small server

Surprising Result

¹The measurements were taken when a cache size of 500K was used, computing the average by accumulating the sum of each time measurement on block removal and dividing it by the number of blocks removed.

cache, but as the cache size is increased the runtime efficiency of the LRU policy, *gained when using a hash table implementation*, plays an important role when deciding on the most efficient cache replacement policy. Although initial results indicated that it is better to cache files, especially when a large cache is used, the average runtime of 16.2 ms for file removal in CFS exceeds both the strategies used by BFS significantly.

Tests on CFS have shown that the network overhead is 3 ms, under a light network load, while under a heavy load it increases to 3.5 ms. Consequently, if a server has a high cache read hit ratio, the data access time for reads via a network can be faster than that of local disk access (11 ms), without a disk cache.

Network Overhead

4.3 Service Analysis

Although a comparison of block versus file cache replacement have been made, the efficiency when reading and writing file data is still a requirement to show overall service performance. This evaluation presents the bandwidth measurement of both services in Section 4.3.1 and concludes by evaluating the response time, under different conditions, of both services in Section 4.3.2.

Section Overview

4.3.1 Bandwidth

The read bandwidth of the block service is calculated when data is read directly from the cache (BFS1) and from physical storage (BFS2). Read bandwidth of the prototype file service is calculated when files are read directly from the cache (CFS1) and from Linux via the cache (CFS2). Write bandwidth of the virtual block storage service is evaluated under three conditions. Firstly, an efficient but less reliable server with a write-behind bitmap block update scheme and a write through data block update (BFS1). Secondly, a write through bitmap and data block update (BFS2) and thirdly, a reliable stable storage server (BFS3) which includes careful storage. BFS4...BFS6 are similar to BFS1...BFS3 (for both reads and writes), with the exception that a block size of 1024 bytes (1K) is used instead of 512 bytes. Table 1 presents the average bandwidth, in kilobytes per second, when reading and writing to both services.

Test Cases

An important observation is that BFS has a much higher bandwidth for both reads and writes, when a block size of 1K is assumed (BFS1...BFS3 versus BFS4...BFS6). This

Bandwidth
Observations

	CFS1	CFS2	BFS1	BFS2	BFS3	BFS4	BFS5	BFS6
Read	362	224	321	230	—	405	315	—
Write	235	—	228	46	5	290	92	11

Table 1: *BFS Average Bandwidth Measured in Kilobytes per second.* This table shows the average read and write bandwidth obtained when both CFS and BFS were tested under certain circumstances. The circumstances are explained in the text.

can be attributed to half the number of writes made to physical storage and half the number of read calls made to the server. When BFS assumes a block size of 512 bytes, the bandwidth of CFS is higher for both reads and writes. When CFS reads a file from Linux, via the cache, the bandwidth decreases considerably from 362 K (CFS1) to 224 K (CFS2) and is significantly less than the BFS bandwidth when reading from disk (BFS5) using 1K blocks as well as using 512 byte blocks (BFS2).

Most of the time only one block update occurs when BFS1 and BFS4 does a write, while BFS2 and BFS5 requires two block updates (one bitmap and one data block). BFS3 and BFS6 requires four updates (same as BFS2 & BFS5 but on two disks) as well as four reads (careful storage write error detection). The effect of disk seek time is shown when the write bandwidth of BFS1...BFS3 are compared with each other, similarly for BFS4...BFS6. The cost of reliability, via stable storage and immediate bitmap block update, is obvious when data is written to the block service. The bandwidth drops from a potential 228 K per second to 5 K per second, for a block size of 512 bytes (BFS1 versus BFS3), and from 290 K to 11 K, for a block size of 1K (BFS4 versus BFS6).

Cost of Reliability

4.3.2 Response Time

Response time measurements indicate how fast a service replies to user requests. An efficient service has a low response time. Response times are measured for CFS by reading and writing files at different sizes. Similar results are generated for BFS using two different block sizes (512 and 1K). The write response time of BFS is generated under three conditions: using a write-behind bitmap block update scheme, updating the bitmap block immediately and using stable storage. These measurements were done when only the measurer used the service. Figure 11 shows the read response time of both services for different file sizes.

Requirement

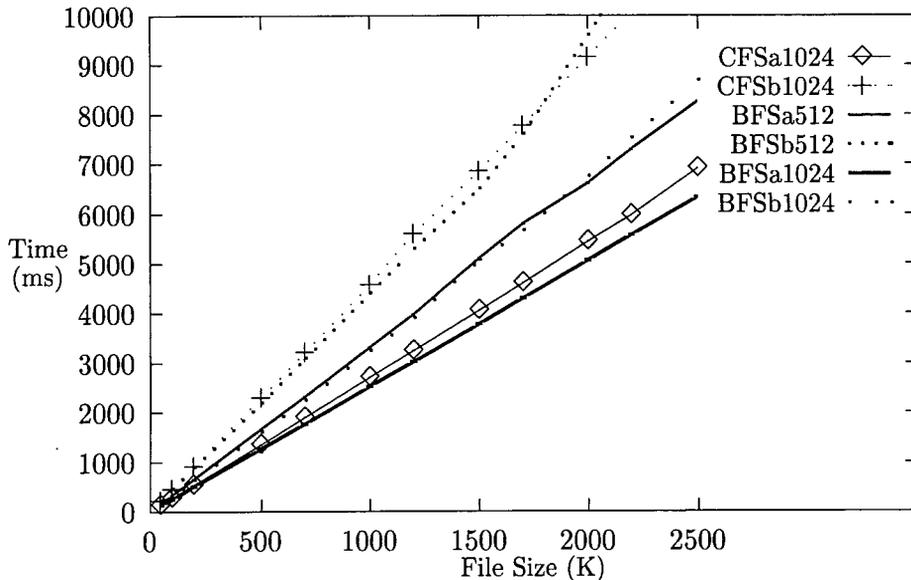


Figure 11: *Response Time of CFS and BFS for Reads.* This graph shows the response time for both services when reading directly from cache (CFSa1024, BFSa512 and BFSa1024) and via physical storage associated with the service (CFSb1024, BFSb512 and BFSb1024). Block sizes of 512 and 1024 bytes are used for BFS.

The measurements show that as the file size increases, the read response time increases linearly. This occurs due to the additional service calls made for larger files. When reading files from CFS via the TcpSVR (CFSb1024) a significant increase in response time occurs compared to reading the file data from its file cache (CFSa1024). This can be attributed to the additional network overhead when reading files via the TcpSVR. The lowest response time for both services occurs when reading from BFS via its cache when using a block size of 1K (BFSa1024). This can be attributed to the additional network overhead when CFS reads from its storage server and the additional overhead in CFS when searching for the file in the users link list of files. On comparing BFS when reading from its block cache at the two different block sizes, the response time difference increases as the file size increases (BFSa512 versus BFSa1024). This occurs because twice as many service calls occur when reading from BFS with a block size of 512 bytes compared to a block size of 1K.

Some Observations

A surprising result from the measurements is that for file sizes above 1.7 MB, reading from disk via BFS using a block size of 512 bytes (BFSb512), a higher response time occurs compared to reading from CFS via the TcpSVR (CFSb1024). This shows the effect

Surprising Results

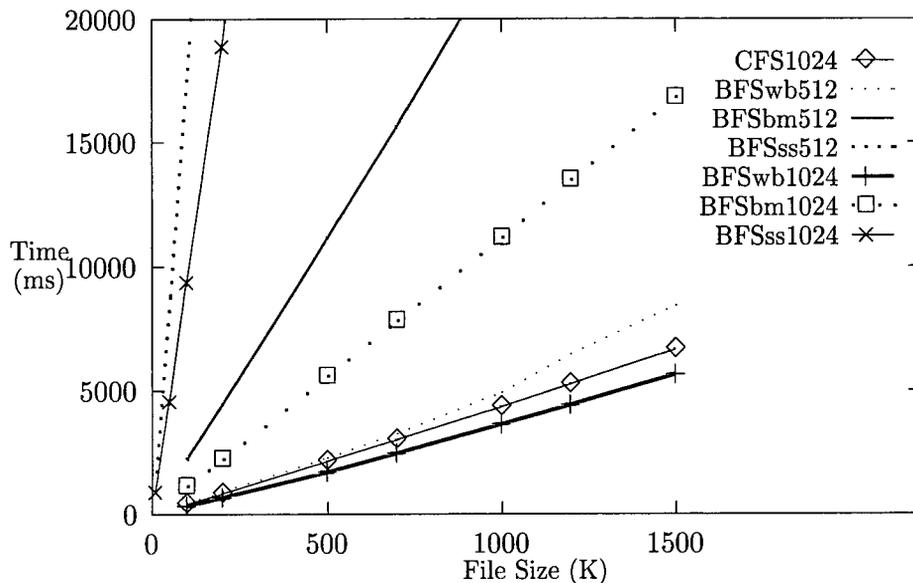


Figure 12: *Response Time of CFS and BFS for Writes.* This graph shows the response time of CFS when files are stored via the TcpSVR (CFS1024). The block service is tested under three conditions. Firstly a write-behind scheme on updating the bitmap block (BFSwb512). Secondly an immediate update of the bitmap block (BFSbm512) and thirdly the stable storage implementation (BFSss512). The graph also shows the response time of BFS when a block size of 1024 bytes is used. (BFSwb1024, BFSbm1024 and BFSss1024).

of the block size when reading larger files from non-volatile storage. Another surprising observation is for file sizes smaller than 2MB: when BFS reads from the cache (BFSa512), a higher response time occurs, when compared to BFS using a block size of 1K when reading from disk (BFSb1024). This can be attributed to twice the number of service calls made to BFS when using a block size of 512 bytes, emphasizing the difference in efficiency when using different block sizes.

Figure 12 shows the response time measurements performed for writes to CFS and BFS under certain conditions. The conditions used for BFS when writes were measured were similar to the conditions (see Section 4.3.1) used for the write bandwidth calculations: a write-behind bitmap block update, an immediate bitmap block update and stable storage update of bitmap- and data blocks. The write response measurements for CFS were done when it updated its file cache and wrote the file data to its storage server (TcpSVR).

	CFS Read	CFS Write	BFS Read	BFS Write Behind	BFS Update Bitmap	BFS SS
Light Load	1.26	1.16	1.26	1.10	1.21	1.16
Medium Load	1.29	1.22	1.32	1.16	1.33	1.25
Heavy Load	1.43	1.30	1.40	1.21	1.41	1.31

Table 2: *Average Response Time Percentage Difference Calculated under different User Loads.* This table shows the average percentage difference from the response time calculated previously. Three different user loads are used: Light, Medium and Heavy.

The write response measurements show that the write-behind scheme for bitmap blocks using a block size of 1K (BFSwb1024) has the lowest response time, lower than writing to CFS (CFS1024). As the file size increases the difference between the write response time of the two services increases. This can be attributed to the additional network overhead when CFS writes to the TcpSVR. The difference in response time when BFS uses a different block size is significant under all three write conditions, especially when the files size is large (for instance BFSbm512 versus BFSbm1024). The response time of writes when BFS uses stable storage is significantly high and becomes unacceptable when large files are written. The cause for this is simple: two block updates (a data- and bitmap block) are required on each disk as well as two confirm reads for the two blocks written for each disk. Recall that the confirm reads are issued by the careful write operation used by stable storage.

Some Observations

Tests on both services have shown that a maximum of 433 read calls per second can occur. This was computed by flooding both services with read service calls, logging the number of calls and dividing it by the time the logging took to complete. In addition to calculating response times for a single user, both services were also profiled under different user file data read loads. Read service calls were chosen to profile the service because of its dominant nature outnumbering write calls. A combination of reads and writes would cause too much inconsistency in the user load to determine response times accurately. Consequently both services were profiled using a light user read load (between 10% and 30% of the 433 calls), a medium user load (between 30% and 60%) and a heavy user load (between 60% and 95%).

User Loads

Table 2 shows the average percentage difference for the three user loads, compared to the response time calculations generated previously under no user load. The user load

User Load

Response Time

difference is more or less equal for BFS when using a block size of 512 bytes or 1K. No significant response time difference occurs. This is because of the high bandwidth and low response time of reads provided by both services. Even a high user load of reads does not influence writing to BFS, using stable storage, significantly.

Chapter 5

Conclusion

The goal of this project was to develop an efficient file storage service. Special attention Project Goal was also given to reliability. A prototype file server was developed first, mainly to provide a much needed service. The first prototype was called CFS. Measurements obtained from CFS guided the design of an improved system which was called BFS. Entire files are cached in CFS, while blocks of fixed size are cached in BFS. The prototype file service CFS has been in use for more than a year and is efficient enough to handle at least 50 clients simultaneously. The results obtained from these two systems are discussed in Chapter 4.

In the improved system BFS each client machine does its own file management while a separate shared server is used to manage disk blocks on behalf of all client machines. A few large disks are therefore shared among many client machines and file management is clearly separated from block management. The block server BFS provides three operations to enable (possibly different) file servers on client machines to manipulate blocks. The block server was designed to be stateless. This simplifies reconnection when a system failure occurs and makes the loss of client data less likely. Greater flexibility is also achieved by providing a separate block service because more than one type of client file system can connect to the block server BFS. The block server also provides protection against some modes of failure by storing redundant information. The results obtained show that it is more effective to cache blocks instead of files under typical working conditions.

In CFS, unnecessary network overhead occurs between the cache (which caches entire Efficiency

files) and the storage manager supported by Linux. The improved system BFS addressed this issue by storing file data on local disks. BFS also uses a more efficient memory management scheme. A separate memory space is maintained for storage blocks and is addressed by using a bitmap. Frequent allocation and deallocation of memory is prevented by allocating the memory (cache) space when the server is initialized. In addition, memory fragmentation is prevented. The results discussed in Chapter 3 show that a more efficient storage service is ensured when caching blocks instead of files and using a write-behind bitmap block update mechanism. Improved response times were obtained as well as higher bandwidth for both reads and writes. This enables more clients to use the service simultaneously.

Since it was a first prototype, reliability was not addressed in CFS. State information is maintained which may be lost during a system failure. The BFS system implements stable storage to address reliability. Although stable storage increases the overhead on writes significantly, reliability is essential in a file server.

Reliability

Another important design issue of any server is availability. The simplicity and stability of the first prototype CFS ensured that availability was never much of an issue in practice. The service can be restarted in a few seconds if a failure occurs although clients must login again after a system failure and they may lose unsaved data.

Availability

The improved system BFS provides a higher degree of availability due to its stateless design. Furthermore, two copies of each disk block are stored on separate disks. Because a system failure may leave the disks in an inconsistent state, a cleanup process is needed. After a system failure, the simplest strategy would be to wait for the cleanup process to terminate before allowing clients to access blocks again. However, this would seriously compromise availability of the service. The solution is to allow clients to start using the system immediately after it has been restarted. The cleanup process is also started, but runs in the background and is interruptable by client requests. Each block is checked for consistency when it is referenced. Although response time is compromised substantially in this way, availability of the service is considered more important.

To compensate for blocks that are not often referenced and that deteriorate over time the cleanup process validates the data storage disk when the service load is low. Therefore the probability of losing client data is low and will only occur when the same blocks on both mirrored devices deteriorate simultaneously.

Comparing the virtual block storage service with the prototype file service showed that a minimum block size of 1K must be assumed. Tests showed that the service was less efficient than CFS, when a block size of 512 bytes was used. Because each partition in the virtual block store caters for different block sizes, a larger block size may be used to improve performance. Block Size

The Future

At the time of writing, the first prototype CFS is still in use. BFS is fully functional, but additional work is necessary before it can replace CFS. Work Required

Security issues should be addressed. Some measures should be taken to protect users against one another. For example, clients should login and provide a valid password before being allowed to use the system. To provide a higher level of security, will require a user service in addition to the block service. Security

The Oberon file system requires a process of garbage collection. Each time a file is modified, a new copy is stored. Old copies are marked as “garbage” and must be removed by a process known as garbage collection. Without garbage collection, the storage space allocated to the Oberon file system on disk will eventually fill up. Garbage Collection

The strategy implemented in CFS was to keep data in the cache as long as possible. Only a “register” operation caused data to be written to disk. Therefore only one copy of each file was stored. A possible solution to be implemented in BFS is to overwrite blocks on disk. A similar strategy to that implemented in CFS would require maintaining extra information about each block, such as whether it has been written to disk or not. This would prevent that a temporary block is removed by a block replacement process. Unfortunately this prohibits the virtual block storage service from running in a stateless environment. If a server crash occurs, clients would therefore lose temporary data.

A possible extension to BFS is to compress data stored on disk. Some research has shown that considerable disk space can be saved when doing ‘on-the-fly’ data compression at block level. Gupta et al. [BGM99] discuss algorithms for doing this in Linux. Compressing data at the file level would lead to a degradation in service performance, especially when updating or reading a small part of the file. The complete file would require compression on each write instead of the relevant data block. As file data is stored on disk, the disk block is compressed. On reading the data from disk the data is uncompressed. To prevent unnecessary uncompression of data each time a client Compressed
Disk Data

reads a data block from a cache, Gupta et al. [BGM99] provides a first level cache for uncompressed data blocks and second level cache that contains compressed data blocks. The second level cache interfaces with non-volatile storage, while the first level cache interfaces with clients.

The experience gained by implementing the prototype file service CFS and the improved system BFS was invaluable. The process of transforming a working system into a usable system proved to be time consuming but well worth the effort. Although I have implemented usable software before, this was the first time that I implemented software of this scale. This convinced me of the necessity for a good design and well defined interfaces between different components of the software system.

Developing Usable
Reactive Systems

Appendix A

CFS Service Commands

Command	Description
Old	Open a file and return a valid file handle.
New	Create a new file and return a valid file handle.
ReadBytes	Read a block and return the data.
WriteBytes	Write a block.
Register	Store file in cache and on physical storage via the TcpSVR.
Search	Search for the existence of a file.
Delete	Remove a file from physical storage and cache.
Rename	Renames a filename in the cache and on physical storage.
Enumerate	Supplies a list of files owned by an user, group or the system wide user. Directory information is attained via the directory cache.

Table 3: *CFS File System Operation Table*. This table shows the operations supplied by the prototype file service and gives a short description of each. The `Purge`, `Unbuffer` and `NewExt` commands, required by the Ceres file system, are not needed due to the design of CFS.

Command	Description	SysAdm
Login	Validate user supplied password and return a valid capability.	
Logout	Logout user.	
Validate	Validate super user status (SysAdm).	*
NewUser	Create a new user in the password cache and file.	*
RemoveUser	Remove user from password cache and file.	*
ChangeUsername	Update new username in password cache and file.	*
ChangePW	Update new password in password cache and file.	
NewGrp	Create a new group.	*
ChangeGrp	Update user group status in group cache and file.	*
ChangeGrpName	Update group name in group cache and file.	*
RemoveGrp	Remove group from the group cache and group file. Ensure that no members are left in this group.	*
RAMFree	Return the current cache memory status.	

Table 4: TcpSVR *User Operation Table*. This table shows the operations supplied by CFS for managing users and groups. In addition to giving a short description of each, SysAdm-only commands are indicated with an asterisk (*).

Appendix B

Disk Storage Service (TcpSVR) Commands

Command	Description
Write	Write a file to disk.
Read	Read a file from disk.
FileInfo	Return file attribute information. This call was used during debugging.
Rename	Rename a file.
Remove	Remove a file or all user files.
DirList	Supply a user directory listing.
Backup	Create a backup of a given file. Used for password and group file.
Restore	Restore old backup of given file. Used for password and group file.
Mkdir	Create a new user.
Rmdir	Remove a user.
Renamedir	Rename a username.
KillSvr	Kill TcpSVR.

Table 5: *TcpSVR Operation Table*. This table shows the operations supplied by the TcpSVR, connected to CFS, and gives a short description of each.

Bibliography

- [Any85] J.A. Anyanwu. A Reliable Stable Storage System for UNIX. *Software: Practice and Experience*, 15(10):973–990, October 1985.
- [BA92] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proceedings of the 12th International Conference in Distributed Computing Systems*, 1992.
- [BEMS91] A. Bhide, E.N. Elnozahy, S.P. Morgan, and A. Siegel. A Comparison of Two Approaches to Build Reliable Distributed File Servers. In *Proceedings of the 11th International Conference in Distributed Computing Systems*, 1991.
- [BGM99] Praveen B., D. Gupta, and R. Moona. Design and Implementation of a File System with On-the-fly Data Compression for GNU/Linux. *Software-Practice and Experience*, 29(10):863–874, October 1999.
- [CDK94] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems – Concepts and Design*. Addison Wesley Publishing Company, Second edition, 1994.
- [Che88] D. Cheriton. *VMTP: Versatile Message Transfer Protocol specification*, February 1988. RFC 1045.
- [Che89] D. Cheriton. VMTP as the Transport Layer for High-Performance Distributed Systems. *IEEE Communications Magazine*, 27(6):37–44, June 1989.
- [Cre94] R.B.J. Crelier. *Seperate Compilation and Module Extension*. PhD thesis, Swiss Federal Institute of Technology, 1994.
- [Dei90] H.M. Deitel. *An Introduction to Operating Systems*. Addison Wesley Publishing Company, Second edition, 1990.

- [DH98] S. Deering and R. Hinden. *Internet Protocol – Version 6 (IPv6) Specification*, December 1998. RFC 1883.
- [DOKT91] F. Douglis, J.K. Ousterhout, M.F. Kaashoek, and A.S. Tannenbaum. A Comparison of Two Distributed Systems: Amoeba and Sprite. *Computing Systems*, pages –, 1991.
- [ED88] R.J. Enbody and H.C. Du. Dynamic Hashing Schemes. *ACM Computing Surveys*, 20(2):85–113, June 1988.
- [FM98] A. Fischer and H. Marais. *The Oberon Companion: A Guide to Using and Programming Oberon System 3*. vdf Hochschulverlag AG an der ETH, 1998.
- [Gra90] C.G. Gray. *Performance and Fault-Tolerance in a Cache for Distributed File Service*. PhD thesis, Stanford University, December 1990.
- [Han98] J. Handy. *The Cache Memory Book*. Academic Press, Second edition, 1998.
- [Jac88] V. Jacobson. Congestion Avoidance and Control. *Computer Communication Review*, 18(4):314–329, August 1988.
- [JL97] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1997.
- [Knu98] D.E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison Wesley Publishing Company, Second edition, 1998.
- [KP87] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *Computer Communication Review*, 17(5):2–7, August 1987.
- [Lam83] B.W. Lampson. Atomic Transactions. In B.W. Lampson, M. Paul, and H.J. Siegert, editors, *Distributed Systems: Architecture and Implementation — An Advanced Course*, pages 248–259. Springer-Verlag, 1983.
- [LS90] E. Levy and A. Silberschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*, 22(4):321–374, December 1990.
- [MdV94] P.J. Muller and P.J.A. de Villiers. Using Oberon to Design a Hierarchy of Extensible Device Drivers. In *Proceedings of the Joint Modular Languages Conference*, September 1994.

- [ME90] D.J. Makaroff and D.L. Eager. Disk Cache Performance for Distributed Systems. In *Proceedings of the 10th International Conference in Distributed Computing Systems*, 1990.
- [MJL84] M.K. McKusick, W.N. Joy, and S.J. Leffler. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [Mul94] P.J. Muller. An Environment for Distributed Programming on a Multicomputer. Master’s thesis, University of Stellenbosch, February 1994.
- [PCBK99] J.S. Plank, Y. Chen, M. Beck, and G. Kingsley. Memory Exclusion: Optimizing the Performance of Checkpointing Systems. *Software–Practice and Experience*, 29(2):125–142, February 1999.
- [PG94] R. Panadawil and A.M. Goscinski. A High Performance and Reliable Distributed File Service. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, 1994.
- [Pos81a] J. Postel. *Internet Protocol*, September 1981. RFC 791.
- [Pos81b] J. Postel. *Transmission Control Protocol*, September 1981. RFC 793.
- [SG94] A. Silberschatz and P.B. Galvin. *Operating System Concepts*. Addison Wesley Publishing Company, Fourth edition, 1994.
- [SKK⁺90] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. Coda: A Highly Available File System. In *IEEE Transactions on Computers*, April 1990.
- [SO90] J.A. Solworth and C.U. Orji. Write-Only Disk Caches. In *ACM SIGMOD International Conference on Management of Data*, 1990.
- [Ste98] W.R. Stevens. *UNIX Network Programming – Networking APIs: Sockets and XTI*. Addison Wesley Publishing Company, Second edition, 1998.
- [TP72] T.J. Teorey and T.B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM*, 15(3):177–184, March 1972.
- [Tso98] Theodore Tso. The EXT2FS Library. July 1998.
- [TvR85] A.S. Tanenbaum and R. van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

- [TW97] A.S. Tanenbaum and A.S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall Software Series, Second edition, 1997.
- [vRvST89] R. van Renesse, H. van Staveren, and A.S. Tanenbaum. The Performance of the Amoeba Distributed Operating System. *Software-Practice and Experience*, 19(3):223–234, March 1989.
- [WEB93] D.L. Willick, D.L. Eager, and R.B. Bunt. Disk Cache Replacement Policies for Network Fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993.
- [Wet98] M.J. De Villiers De Wet. A Development Environment For Reactive Systems. Master's thesis, University of Stellenbosch, December 1998.
- [WG92] N. Wirth and J. Gutknecht. *Project Oberon—The Design of an Operating System and Compiler*. Addison Wesley Publishing Company, 1992.
- [WHK97] Y.M. Wang, Y. Huang, and C. Kintala. Progressive Retry for Software Failure Recovery in Message-Passing Applications. In *IEEE Transactions on Computers*, 1997.