Design and Implementation of a Configurable Mixed-Media Filesystem
EXTENDED ABSTRACT

*Silvano Maffeis**
Computer Science Department, University of Zurich, Switzerland
`maffeis@ifi.unizh.ch`

IFI TR 94.03

June 1993

Abstract

Traditional UNIX[1] filesystems are not adequate enough for directly supporting database systems, multimedia systems or applications requiring high I/O performance. In this paper we describe the design and implementation of a configurable mixed-media filesystem. The attribute *configurable* means that a filesystem serving a specific application area can be realized with less effort out of a library of reusable filesystem classes. The attribute *mixed-media* stands for the filesystem's ability to integrate different media types (RAM, harddisks, WORM optical disks, CDROMs, tape devices, RAIDs etc.) into a virtual storage and making applications unaware of this aggregation. A prototype C++ implementation of the proposed design, called the VANILLA filesystem, is presented and its performance assessed. Raw-write performance is up to 4.5 times higher than in a standard Sun OS filesystem. We will also demonstrate how various storage organization forms, especially hierarchies, arrays, and mirrors of both local and remote storages, can be realized using an expressive syntax.

1 Problem Statement

Traditional UNIX filesystems [8] were designed to support computers with little RAM and small disk devices. Files are split into fixed size logical blocks and sequentially accessing a whole file induces several movements of the read/write head often causing long delays. This scheme is inefficient, since, as the well-known study conducted by Ousterhout and others [10] reports, more than two thirds of the file accesses in an academic UNIX environment are whole-file transfers. Better performance can be achieved by maintaining files *contigously* on storage [15, 13, 4]. Other drawbacks of the traditional UNIX filesystem design are:

- the difficulty to integrate different media types, like RAM, harddisks and WORM optical disks to a single, virtual filesystem. A novel approach to such a filesystem is described in [12].
- its inadequateness for database applications [9, 3]. Database systems for UNIX, like Oracle or Exodus, circumvent the UNIX filesystem by directly implementing their own storage system on the *raw devices*, which makes UNIX database systems more expensive.
- its inadequateness for multimedia applications. Multimedia applications require high I/O throughput rates and quality of service guarantees like constant minimum data rates [1].

*This work is supported by *Siemens AG ZFE, Germany*, by the *Union Bank of Switzerland*, and by the *Eidgenössisches Volkswirtschaftsdepartement, Kommission zur Förderung der wissenschaftlichen Forschung (KWF), Switzerland*, Grants No. 2255.1, 2554.1

[1] UNIX is a registered trademark of Unix Systems Laboratories, Inc.

- the low raw throughput. Traditional UNIX filesystems do not scale well with the ongoing increase in CPU performance. A filesystem supporting datastriping [5] and diskarrays would help in overcoming the I/O bottleneck.
- the weak configurability. E.g. it is difficult to change the buffer replacement policy (LRU) of the UNIX filesystem. In particular, LRU performs marginally for many database [9] and multimedia applications.

In this paper we present the design and implementation of a *configurable, mixed-media filesystem.* Configurable in the sense that features like the file replacement policy [6], the space-allocation algorithm and such like can easily be changed and that a variety of storage organization forms, like replicated storages, storage hierarchies, and striped storages can be configured using an expressive syntax. The attribute *mixed-media* indicates that virtual storage systems comprising RAM-cache, harddisks, WORM optical disks, tape devices etc. can be configured. We have implemented a prototype UNIX filesystem, called the VANILLA filesystem (VFS), in C++ [14] following the design presented in the next sections. VFS allows to plug together an arbitrary number of storage devices into a virtual storage, making applications unaware of this aggregation.

2 Design and Implementation of the Vanilla Filesystem

2.1 Design Philosophy

To solve the problems we stated in section 1, a generic filesystem design in form of a C++ class library is proposed. Filesystems serving a specific application area, e.g. database or multimedia applications, can be constructed by reusing and adapting these classes. Class inheritance and method overwriting allow the tailoring of the filesystem classes to special needs without having to implement a special filesystem from scratch.

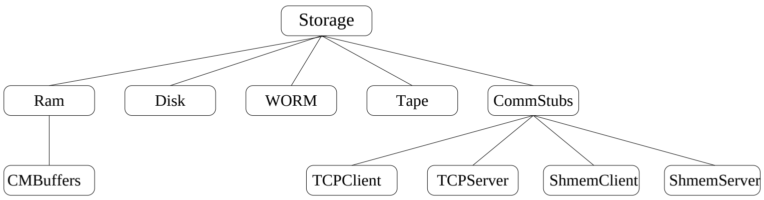

Figure 1: Part of the Vanilla FS class hierarchy.

Figure 1 depicts the relevant parts of the proposed class hierarchy[2]. The key idea is that most of the required program code, for example the free list management algorithms, the replacement policies, code supporting disk arrays and replication, is contained in a generic, device and operating system independent base class Storage, and can be reused by the derived, device-dependent classes. More than 80% of VFS are realized in the device and operating system independent base class Storage.

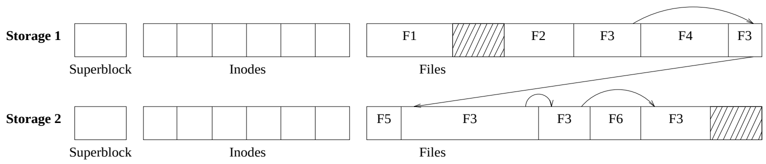

Figure 2: Storage layout of the Vanilla FS. File F3 is fragmented over two storages.

The storage layout is shown in figure 2. The default is to store files contigously. This has the advantage that entire files stored on disk can efficiently be accessed with a minimum of movements of the read/write arm. Another important advantage is that this layout is also suited for building efficient filesystems on serial

[2] Class CMBuffers realizes real-time storage and retrieval of continuous media data [1]. For space reasons it is not addressed in this abstract.

access devices (like tape devices), on write-once devices (like WORM disks), and on random access devices (like RAM-caches) at the same time.

Very large files and multimedia data streams are stored using *fragmented files*. Fragmented files are formed out of a chain of interconnected clusters, like file F3 in figure 2. The fragment size is variable and can be chosen by the programmer on a per-file basis.

2.2 Device Dependent Subclasses

Device dependent subclasses, like Ram, Disk or WORM (see figure 1) only need to provide a small set of device dependent methods. For example, the Disk class is implemented directly on the UNIX *raw devices*. Implementing a device dependent storage class mostly consists in realizing three simple virtual methods:

- pmove() performs an overlapping move of storage regions. This virtual method is called from within the code of the Storage class to compact a real storage. As can be seen in figure 2, a gap in the storage results when a file is deleted. Thus, the storage will become more and more fragmented during its operation and compaction of the storage consists in moving allocated chunks to the top of the storage.
- read() reads a specific amount of data and
- write() writes a specific amount of data.

2.3 Some Example File System Configurations

The following code fragment demonstrates a hierarchical file system configuration using the VFS class library. File manipulation operations are always issued on the "topmost" storage. When this storage becomes full, the filesystem automatically moves files to the next attached storage until enough free space has been created to satisfy the request. The replacement policy (LRU, MRU, LFU or MFU) decides which files to replace. Dependent on the situation specific file access patterns [7], programmers can provide their own replacement policies by overwriting one specific method of class Storage.

```
Ram cache;
Disk disk("/dev/rsd1g");

cache.install(4194304, 200);          // 4mb cache, 200 inodes.
disk.install(10000);                  // 10000 inodes.

cache.attach(disk);                   // configure the filesystem (hierarchy).

File f = cache.create();              // some simple file operations:

cache.write(f, buf, n);
cache.read(f, buf, n);
cache.close(f);
cache.unlink(f);
```

In the next example, we demonstrate how a mixed-media filesystem consisting of two disk partitions, a RAM-cache and a WORM optical disk can be plugged together:

```
Ram cache;
Disk disk1("/dev/rsd1g");
Disk disk2("/dev/rsd1h");
Worm worm("/dev/worm");
```

```
cache.install(4194304, 200);                     // 4mb cache, 200 inodes.
disk1.install(10000);                            // 10000 inodes.
disk2.install(10000);                            // 10000 inodes.
worm.install(100000);                            // 100000 inodes.

cache.attach(disk1).attach(disk2).attach(worm);  // a cached WORM filesystem.
```

Several data and parity disks can be configured to form a *redundant array of inexpensive disks (RAID)* [2] system. The striping unit [11] is variable and can be chosen on a per file basis:

```
Disk disk1("/dev/rsd1g");
Disk disk2("/dev/rsd2g");
Disk disk3("/dev/rsd3g");
Disk disk4("/dev/rsd4g");
Disk parity("/dev/rsd5g");

disk1.install(10000);
disk2.install(10000);
disk3.install(10000);
disk4.install(10000);
parity.install(10000);

disk1.stripe(disk2).stripe(disk3).stripe(disk4);  // configure the disk array.
disk1.add_parity_device(parity);                  // like a RAID 3 system.

File f = disk1.create(512);                       // striping unit = 512 bytes.

disk1.write(f, buf, n);
```

The I/O to a storage array can be *parallelized* by having an independent worker processes per disk. A coordinator process communicates with the worker processes using the shared memory communication stubs ShmemClient and ShmemServer depicted in figure 1. In analogy, remote devices can be accessed using plug-compatible communication stub classes like TCPClient and TCPServer:

```
// ON THE CLIENT SIDE:
Ram cache;
TCPClient client("eterna.ifi.unizh.ch", 9000); // host and TCP port number.

cache.install(4194304, 200);
client.connect();                              // connect to the remote stub.

cache.attach(client);                          // a cached network filesystem.

File f = cache.create();                       // write on the remote FS.
cache.write(f, buf, n);
...

// ON THE SERVER SIDE:
Disk disk("/dev/rsd1g");                       // the real device.
TCPServer server(9000);                        // server stub.
```

```
disk.install(10000);
server.attach(disk);                    // configure the filesystem.
server.listen();                        // listen on port.
```

3 Naming and Directory Serving

VFS uses unique object identifiers to identify files. To access files using a more user friendly naming mechanism, e.g. hierarchical UNIX paths, special *name servers* can be instantiated and used to manage a storage:

```
UNIXNameserver ns;
Disk disk("/dev/rsd1h");
...

ns.administer(disk);

File f = ns.open("/tmp/test", O_CREAT, 0644);
ns.write(f, buf, n);
...
```

Storage management and name serving are separated for flexibility. Programmers can provide their own naming mechanisms easily without having to modify the kernel or the filesystem code. Different nameservers can be used to administer the same storage if required. Nameservers are realized as a separate class hierarchy[3].

4 Performance Considerations

In this section we present a performance comparison between VFS and an out-of-the-box SUN OS 4.1.3 filesystem. All measurements were performed using a Seagate ST42400N harddisk attached to a SUN SPARCstation 2 workstation. Figure 3 compares the *raw, unbuffered throughput* of write operations. The amount of

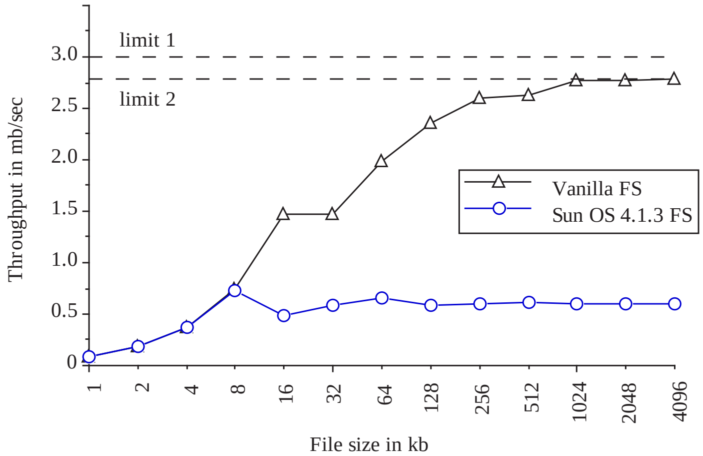

Figure 3: Comparison of raw write performance.

data written ranged from 1 kb to 4 mb. The results show that VFS allows to use more than 90% of the device's maximal raw throughput, which is 3mb/sec (limit 1) for the Seagate ST42400N harddisk[4]. For large

[3]Not illustrated in this abstract for space reasons.
[4]Declaration of the manufacturer.

files, more than 99% of the maximal available raw device throughput (limit 2) can be used[5], whereas the Sun OS filesystem can use at most 22% of it. For large files, VFS' raw throughput is 4.5 times higher than in the Sun OS filesystem. The proposed file system design achieves both high flexibility and high performance.

[5] Other measurements not included in this abstract show that for file creation-write-unlink operations VFS is 2 to 7 times faster than the Sun OS filesystem, even for files smaller than 8 kb.

References

[1] Anderson, D. P., and Osawa, Y. A File System for Continuous Media. *ACM Transactions on Computer Systems 10*, 4 (Nov. 1992).

[2] David A. Patterson and Peter Chen and Randy H. Katz. Introduction to Redundant Arrays of Inexpensive Disks (RAID). In *COMPCON, 34th IEEE Computer Society International Conference* (Feb. 1989).

[3] J. Eliot B. Moss. Getting the Operating System Out of the Way. *IEEE Database Engineering 9*, 3 (Sept. 1986).

[4] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proceedings of the Winter 1991 USENIX Conference* (Dallas, TX, Jan. 1991).

[5] Luis-Felipe Cabrera and Darrell D. E. Long. Exploiting Multiple I/O Streams to Provide High Data-Rates. In *Proceedings of the Summer 1991 USENIX Technical Conference and Exhibition* (Nashville, Tennessee, June 1991).

[6] Maffeis, S. Cache Management Algorithms for Flexible Filesystems. *ACM SIGMETRICS Performance Evaluation Review 21*, 2 (Dec. 1993).

[7] Maffeis, S. File Access Patterns in Public FTP Archives and an Index for Locality of Reference. *ACM SIGMETRICS Performance Evaluation Review 20*, 3 (Mar. 1993).

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

[9] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM* (July 1981).

[10] Ousterhout, J. K., et al. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th Symposium on Operating System Principles* (Dec. 1985).

[11] Peter M. Chen and David A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (May 1990), IEEE.

[12] Quinlan, S. A Cached WORM File System. *Software Practice and Experience 21*, 12 (Dec. 1991).

[13] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992).

[14] Stroustrup, B. *The C++ Programming Language.* Addison Wesley, 1986.

[15] van Renesse, R., Tanenbaum, A. S., and Wilschut, A. The Design of a High-Performance File Server. In *9th International Conference on Distributed Computing Systems* (Newport Beach, CA, 1989), IEEE.