PFS: A Distributed and Customizable File System

Peter Bosch*
CWI
Netherlands
peterb@cwi.nl

Sape Mullender
University of Twente
Netherlands
sape@cs.utwente.nl

Abstract

In this paper we present our ongoing work on the Pegasus File System (PFS), a distributed and customizable file system that can be used for off-line file system experiments and on-line file system storage. PFS is best described as an object-oriented component library from which either a true file system or a file-system simulator can be constructed. Each of the components in the library is easily replaced by another implementation to accommodate a wide range of applications.

1. Introduction

We have built an object-oriented and distributed file system that can be employed as a true distributed file system, but can also be used as a file-system simulator to perform off-line[1] file-system experiments. This work started several years ago as a project to build a system that can handle ordinary file data as well as multi-media data. It evolved to an object-oriented distributed file system and simulator for a number of reasons. This paper presents those reasons and describes the lessons we have learned while building the system. We feel that these lessons are generally applicable to other operating system work.

When we started out with the file-system project, we first tried to realize our ideas in existing file systems. At that time most file systems that were in use were large monolithic blocks of code, usually embedded in the operating system. The disadvantage of a single monolithic file system is that it is hard to change. Many implicit policy decisions are hidden somewhere in the (seemingly) unstructured code. We found it a non-trivial task to find them and to change them to do something different.

To build efficient storage algorithms for new application areas, it is indispensable to perform performance experiments. Usually, system bottlenecks occur at unexpected places and to find those places detailed analysis of the system is required.

If the system is one monolithic block and tightly bound into the native operating system it may be frustrating to perform such experiments. Often it is impossible to lift the system from the surrounding operating system and to run it stand-alone on a workbench. If one has succeeded in separating the system from its surroundings, the system is usually still a monolithic block of code. Learning the internals of the system is still hard. We think this is wrong: in our opinion it must be possible to run the system on a workbench and to open the system's internals for inspection.

Another approach to execute performance experiments is to build approximations of the real system to answer performance-related questions. We think this approach is wrong. Too much detail may be lost when performing I/O experiments and one may be measuring things that in a real system do not turn out to be bottlenecks. Ruemmler et al. [15] measured performance differences of up to 112% in their disk simulator when they were modelling and implementing simulated disks. For the initial versions they did not implement parts of the disk, which turned out to be important for overall disk performance. We have had a similar experience with an initial file-system simulator. By not modelling file-system meta-data updates for each read operation, simulated performance did not reflect real performance.

We believe that the only way to build a realistic simulator is to run exactly the same code in a the simulator as in the real system.

After our unsuccessful attempts to add our ideas to existing file systems, we built our own system. From the beginning we kept in mind that the system would be used for experimentation: it had to be written such that we could change the system easily. We also found it important to be able to re-use earlier written code.

We split the system in several key components and im-

*Usual affiliation: University of Twente, Netherlands.

This work is supported by the PEGASUS project (ESPRIT BRA 6586) at the University of Twente.

[1] Off-line here means "not for on-line storage".

1063-5351/96 $5.00 © 1996 IEEE
Proceedings of IWOOOS '96

plemented those components in a set of C++ base classes, which implement a basic file-system. The basic file system implements a Unix file system with a Log-structured File System (LFS) [14, 16] back-end. Additional policies are added by extending and partly overloading the base classes with new policies.

Writing the system in an object-oriented manner has proven to be quite useful for two reasons. First, testing new algorithms and policy decisions required only the addition of a new subclass and possibly rewriting some of the superclass' methods. Second, we were forced to make a split between policy and mechanism. If mechanism and policy are still integrated, testing new policies implies implementing the whole mechanism over and over again. Note that for many parts of the system we did not get this "right" the first time. Rewriting parts of the system to get the split right gave us a better understanding of the underlying system issues. Our approach resembles the Choices approach [4] although we split our system such that it is quite simple to change internal file-system policies.

We were faced with the problem of how to run the system on a workbench. To solve this we used a technique similar to the one used by Thekkath et al. [17]. In that system a standard Unix file system was lifted from the Unix kernel and run as a Unix user process. Additional *helper* components simulated the surrounding operating system. We used a similar trick: we implemented a set of C++ classes that simulate the behaviour of the surrounding system (e.g. processor, memory, I/O hardware, time) and we were able to run our file system as an off-line simulator.

We found an additional problem if one is using an extensible and customizable file system and simulator. Each of the file-system experiments generates a wide range of performance results that are only valid for some configuration of the system. In order to keep track which configuration created what results, we are currently designing an experimentation database. Such a database holds all system components, performance graphs, system workloads, and file-system *snapshots* from all experiments: it will always be possible to revert to an earlier experiment or system[2].

There exist several other simulation environments. SimOS is a complete machine environment that simulates CPUs, caches, memory systems and a number of I/O devices [13]. SimOS is used to run a full (and unmodified) operating system in a simulated hardware environment. This approach is similar to Thekkath's approach to run existing file systems in simulators. The difference with our approach is that we also supply a workbench for file-system experimentation rather than a tool for system measurements.

[2]For this work, we have defined file-system snapshots as complete dumps of a file-system's meta data (super-blocks, inodes, and directory contents). They usually serve as a starting point for trace driven analyses.

The only system to our knowledge that resembles our system is the Pantheon disk simulator [18, 15, 5]. In Pantheon complete disks (controllers, caches, internal queues) can be built and measured. It is mainly used to design new storage systems or to find bottlenecks in existing systems.

The remainder of this position paper is organized as follows. In Section 2 we describe our system in some detail. In Section 3 we describe some of the experiments and systems we have built or are planning to build. Section 4 contains concluding remarks.

2. Pegasus File Server (PFS)

Figure 1 shows our basic system configuration. The Pegasus File System (PFS) consists of a file system, a file-system simulator and an experimentation database. The file system is used for ordinary file storage, the simulator is used for workbench file-system experiments and the database is used as an aid for the simulator and file system.

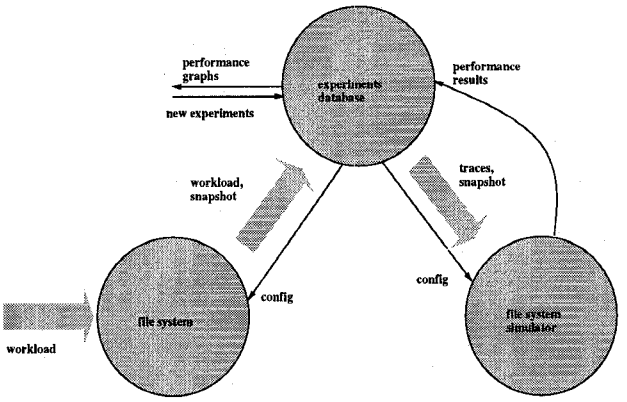

Figure 1. System configuration.

The database is the central point in our system. It holds all components to construct file systems and file-system simulators, it maintains file-system snapshots, keeps earlier recorded workloads for playback on simulators together with the system configuration on which they were created and it maintains performance results of earlier experiments.

File systems and simulators are checked out from the database by specifying a specific configuration to the database. The experimentation database returns which file-system components are required to build the system (each of the components is implemented by a C++ class).

There are components that implement caches, file-system front-ends, disk layouts and disk driver interfaces. Simulator components usually implement hardware components in software. There are, for example, components that implement disk drives, SCSI busses and tape devices and we are working on components that implement various

types of networks. Finally, there are helper components that provide the glue between the file-system components and the underlying operating system.

When a file-system simulator is built, it can be handed a snapshot file system. This snapshot serves as a starting point for simulations and allows the playback of earlier recorded workload. The database runs the simulation by sending earlier recorded workloads to the simulator. The simulator maintains performance statistics and when the simulation finishes, the simulator installs the performance results in the database for further analysis.

Snapshots and workloads are preferably created by the file system itself. The file system makes a snapshot of all the meta-data in its local file-system disks, sends the snapshot to the database and starts recording all operations that are executed on the file system. The advantage is that in this case the exact configuration of the system is known and can be rebuilt for experimentation.

It is also possible to download other file-system traces in the database (e.g. the Sprite [1] or Coda traces [10]). The disadvantage of pre-recorded traces is that it is usually harder to get access to snapshots and system configurations. It is not unusual that the system configuration is hardly known anymore [7] and needs to be synthesized from the traces. Nonetheless, these traces are still important because they represent a true system workload.

Possible configuration

Figure 2 shows a possible configuration of our system (in fact, it is the system we are currently working on). This system consists of a client/server configuration. The client configuration is bound through the front-end to the user's workstation. The server is responsible for maintaining the actual file system on disk on request of the clients.

Client requests that arrive at the client agent are managed by the system as follows. First, they are dispatched to a set of internal file-type specific services. These file dependent services (e.g. directory read), map the requests onto more fundamental file-system cache requests. When a client request needs servicing by the central server, the caches forward the request to the central server through a remote file system component. The server receives the client request through the server channel and maps the calls onto a similar stack of modules. If a request needs servicing by the disk, the request is dispatched to the file-system layout module, which associates actual sector numbers to the request. The file system module dispatches the request to the I/O scheduling module to read or write data from disk.

In the simulator case, hardware is modeled and implemented by simulator components that behave exactly as their real counterpart. All simulated hardware delays the system for exactly the same amount of time as their real

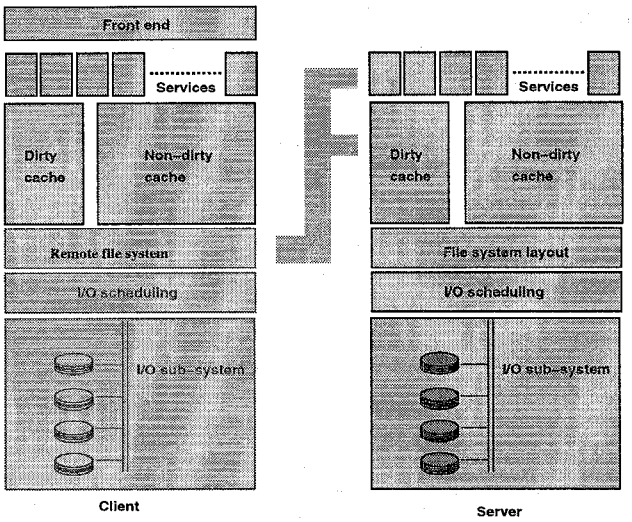

Figure 2. All file-system components.

counterpart would do. The simulated components also match the semantics of the real devices.

We model hardware much like disks in HP's Pantheon disk simulator [18, 15, 5]. A "software" disk is first implemented through the factory specifications, and calibrated by comparing true performance to simulated performance. Basically, our disk models are re-implementations of the Pantheon disk models. In the future we hope to glue the Pantheon and our file-system simulator together.

In our system, simulated hardware can also deal with real data. For this, we download a file-system snapshot from the experimentation database into the simulator before the simulation starts. The simulated hardware reads and writes data from and to the snapshot when the real file-system components would request data from the disk. For reasons of privacy and manageability we only use the file-system meta-data (including directories) for snapshots. Ordinary file-system data is initialized to zero.

The performance of the file-system simulator is similar to the performance of the file server since the simulator is executing the same operations as the file server. In the case that an idle period is encountered, the simulator simply steps over the idle time, while the real system needs to wait for the next operation. However, simulated hardware usually performs slower than real hardware devices (especially if the system handles true data). We do not consider the performance of the simulator as a real issue because we run all our simulations as batch jobs. A real benefit from our approach is that we can measure the performance of a file system without purchasing the hardware.

Current state

The current state of the system is that both file system and file-system simulator exist, are fully functional and are used for experiments. Both systems derive their core system from a CVS source tree to construct a system binary. Both systems are fed a configuration file to construct a system at startup. Once a system has been built, it is not possible anymore to change the internal components.

The current experiments database consists of a set of utilities to keep track of earlier generated performance tables, earlier snapshots, recorded traces and the separate CVS source tree. The problem with the current set of tools is that they are not yet integrated: it is quite simple to loose earlier configurations and performance results.

We are currently replacing the set of utilities by one simulation management tool: the experimentation database. This database is able to extract sources from the CVS tree, combine them to a runnable system, to run the system and to collect the generated performance results in a database (combined with the system that generated the results) for later examination. Please note that this experimentation database does not exist yet.

3. Experiments and systems

We have performed various file-system experiments in our system. In this section we summarize the most important versions of our system.

Delayed updates

The most important experiment we have conducted so far are delayed update experiments. In these experiments, we delay disk write operations for longer periods in the hope that new writes overwrite old writes. This is particularly important for a Unix workload as this workload is characterized by a high overwrite and discard factor [11, 1, 8]. In Unix, many files are discarded quickly from the file system and completely re-written. If we avoid writing those files to disk, we can minimize disk write operations, which reduces disk queues and I/O times.

For this experiment we re-constructed the Sprite system that was used for the file system measurements as reported in the 1991 SOSP [1] and we re-ran the Sprite traces on the system.

We found that delaying disk write operations for a longer period increases read cache misses, but also reduces the average length of the I/O queues and improves overall file-system performance [2, 3]. Our results are in contradiction with Chen's work [6], which shows an "optimal" delay of 1–10 seconds for a Unix-like workload. We found different results because we assume that for most workloads, the file-system cache can easily accommodate write bursts. This means that we can focus on file-system read and write latencies. It turns out that it is important to delay writes to reduce disk read/write and write/write contention. Chen's system does not address these issues.

In order to guarantee data persistency we came up with a distributed update protocol, which resembles Harp's distributed update protocol [9]. The basic idea of this protocol is that as long as data is volatile, it is protected by maintaining two copies of the volatile data in two machines (one in the client machine, one in the server machine). If only one of the two machines fails, the other machine still owns a copy of the data.

We extended our on-line file system with the distributed update protocol and we ran some initial performance measurements. It showed us that delaying writes improves file-system performance. We are currently fine tuning the system and we will report on the overall performance separately[3].

Ordinary file storage

We are using the basic file system configuration for ordinary file storage. The system is mounted through NFS to our Unix name space and the system provides ordinary Unix-like file I/O.

Our plan is to use the basic file system configuration as a system to create file-system traces on. Every once in a while all file I/O operations are recorded in the experimentation database, which enables us to replay parts of true workload on the simulator workbench.

Multimedia experiments

We have used our file system for a multi-media experiment on Nemesis, a multi-media kernel [12]. We ported our file system to this micro-kernel and we added components that deal with large multi-media files, a new caching algorithm for multi-media files and a new instance of our file-system layout module. The remaining parts of the system were left unchanged.

The ease with which we introduced a fundamentally new file type to our file system showed us that our approach is worthwhile. We did not have to build a complete new file system for the Nemesis micro-kernel, we only had to add functionality that deals with the new file type. We were able to concentrate on multi-media storage issues alone and we did not have to bother about many other parts of the file system.

[3] A full analysis is beyond the scope of this paper.

4. Concluding remarks

We have presented an object-oriented and customizable file system that can be used for true file storage and file-system simulations. It allows us to perform file-system experiments off-line on a workbench and when we are satisfied with the system's performance, we can migrate the analyzed file-system algorithms to a real file system.

By using an experimentation database we will be able to store intermediate simulation results, snapshots of file-systems, workloads executed on the file systems for later playback, and file-system configuration information. It will help us to keep track of earlier experiments and configurations. We think this is important for later re-validation of results and to compare new algorithms to old ones.

We have found this system quite useful: new systems are easily built and tested. By carefully splitting mechanisms and policy in the component library, we can easily test new policies without having to rewrite large bodies of code.

We feel that our approach is generally applicable in other areas of operating system research. In particular, the experimentation database and the helper components can easily be used for other types of operating system experiments.

Acknowledgements

We thank Richard Golding, HP Labs for initiating our work on the experimentation database and for proofreading earlier versions of this paper. We also thank the anonymous reviewers for their comments.

References

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

[2] Peter Bosch. A cache odyssey. M.Sc. thesis, published as Technical Report SPA–94–10. Faculty of Computer Science/SPA, Universiteit Twente, the Netherlands, 23 June 1994.

[3] Peter Bosch and Sape J. Mullender. Cut-and-paste file-systems: integrating simulators and file-systems. *USENIX* (San Diego, CA), January 1996.

[4] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. Choices, frameworks, and refinement. *Proceedings of Workshop on Object Orientation in Operating Systems* (Palo Alto, CA), pages 9–15. IEEE Computer Society Press, October 1991.

[5] Pei Cao, Swee Boon Lin, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP Parallel RAID Architecture. *ACM Transactions On Computer Systems*, **12**(3):236–69, August 1994.

[6] Peter M. Chen. Optimizing Delay in Delayed-Write File Systems. Technical report CSE-TR-293-96. University of Michigan, 1996.

[7] John H. Hartman. Allspice's configuration, 24 March 1995. Private communication.

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

[9] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. Technical report MIT/LCS/TM-456. Mass. Institute of Technology, Laboratory for Computer Sc., Cambridge, MA (USA), August 1991.

[10] L. Mummert and M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. CMU-CS-94-213, November 1994.

[11] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[12] Timothy Roscoe. Linkage in the Nemesis single address space operating system. *Operating Systems Review*, **28**(4):48–55, October 1994.

[13] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. *Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Association for Computing Machinery SIGOPS, 3–6 December 1995.

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

[15] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, pages 17–28, March 1994.

[16] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 307–26. USENIX, Winter 1993.

[17] Chandramohan A. Thekkath, John Wilkes, and Edward D. Lazowska. Techniques for file system simulation. Technical report HPL–OSR–92–8. HP Labs, October 1992.

[18] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *Proceedings of 15th Symposium on Operating System Principles*, 1995.