The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Cut-and-Paste File-Systems: Integrating Simulators and File-Systems

Peter Bosch and Sape J. Mullende
Universiteit Twente

Cut-and-Paste file-systems: integrating simulators and file-systems

Peter Bosch, Sape J. Mullender
Faculty of Computer Science/SPA, University of Twente, Netherlands
peterb@cs.utwente.nl, sape@cs.utwente.nl

Abstract

We have implemented an integrated and configurable file system called the Pegasus file-system (PFS) and a trace-driven file-system simulator called Patsy. Patsy is used for off-line analysis of file-system algorithms, PFS is used for on-line file-system data storage. Algorithms are first analyzed in Patsy and when we are satisfied with the performance results, migrated into PFS for on-line usage. Since Patsy and PFS are derived from a common *cut-and-paste* file-system framework, this migration proceeds smoothly.

We have found this integration quite useful: algorithm bottlenecks have been found through Patsy that could have led to performance degradations in PFS. Off-line simulators are simpler to analyze compared to on-line file-systems because a work load can repeatedly be replayed on the same off-line simulator. This is almost impossible in on-line file-systems since it is hard to provide similar conditions for each experiment run. Since simulator and file-system are integrated (hence, use the same code), experiment results from the simulator have relevance in the real system.

This paper describes the cut-and-paste framework, the *instantiation* of the framework to PFS and Patsy and finally, some of the experiments we conducted in Patsy.

1 Introduction

Building a fast file-system or introducing new algorithms to an existing file-system is often a difficult task. The designer does not know in full detail what a proposed algorithm does to the overall file-system performance. Normally, the only way to test a new algorithm is to introduce it to an on-line file-system and measure the performance effects while the system is in use. If a new algorithm does not work, file-system service may be interrupted.

A better way to test algorithms is to analyze the performance of those algorithms in an off-line simulator. When the algorithm works as expected, it can be integrated in a production file-system. However, the disadvantage of an off-line simulator is that it is hard to build a representative simulator. Usually, simulators are approximations of real systems [21], and results from such a simulator may not show actual system performance.

Recent developments [20, 26, 27, 3, 8] in file-system research have shown us that improving file-system performance is still a hot topic. Many groups find new storage algorithms and report better performance numbers. These performance numbers usually show the effectiveness of those algorithms within the environment they were developed for.

Ideally, to validate those reported performance numbers, other researchers would re-use (parts of) the presented system, insert it into their own environments, and re-evaluate the published performance. In reality, this is well nigh impossible [5]. Often the system code heavily depends on the environment for which it was developed. Through some effort it is possible to port the source code, but then it is hard to set up a similar test environment [9, 29].

To solve both problems, we have designed and implemented an *extensible reference file-system component library* from which we instantiate on-line file-systems and off-line file-system simulators. On-line systems are systems that are in use in real systems, off-line systems are not in use by clients and only run in a controlled environment[1]. The component library provides basic components that are required to build a full file-system and the same components are used to build file system simulators. *Helper* components are added to complete an instantiation to a real file-system or simulator. Helper components in a real

This work is supported by the PEGASUS project (ESPRIT BRA 6586) and the BROADCAST project (ESPRIT BRA 6360).

[1] We have not yet experimented with off-line file-systems.

system deal with actual data movement, while in the simulator they compensate for the lack of "real" data. We have made the component library such that it is easy to extend the library with new algorithms and policies.

The symbiosis of the simulator and the "real thing" helps us to:

- be more confident that simulated off-line performance numbers show real and representative on-line file-system performance numbers;
- easily detect performance bottlenecks of real file-system algorithms by running the same system off-line in an *equivalent* file-system simulator;
- analyze new file-system algorithms off-line before they are integrated into a production file-system;
- be more confident that no side effects are introduced when a simulated algorithm is moved into a real system;
- construct a reference file system: other file system algorithms can easily be integrated and compared to other algorithms in our environment;
- easily migrate algorithms from off-line simulators in real systems: the algorithm does not have to be re-implemented for the real system once implemented for a simulator.

Our initial goal for this work was to build a production file-system and a separate file-system simulator. The production file-system was to be used for ordinary data storage with functionality to store continuous media files, the simulator was to be used for analyzing storage algorithms.

In the original simulator, we analyzed cache-flush algorithms. For that work we replaced the Unix 30-second-update timer policy by a flush policy that keeps dirty data in the cache much longer [4], so-called *write-saving* policies. Unix file-system write traffic is characterized by a high overwrite factor in the first part of a file's lifetime [2, 10, 17, 22]. Keeping dirty data longer in memory without writing dirty data to disk increases the probability that a block is overwritten through *truncate* and *delete* calls in memory rather than on disk. As a result fewer data blocks are written to disk.

The work showed that replacing the 30-second-update timer by a write-saving policy greatly reduces file-system read latencies. The work claims that disk I/O queues are the main cause of relatively high file-system latencies. Basically, the goal was to get *writes out of the way of reads* simply by writing less data to disk.

The initial simulator used for the experiments used an approximation of a real file system and it used a simple disk model. As is shown by Ruemmler et al. [21], a simple disk model in a simulator may not show the actual performance: the results can be completely useless. Ruemmler et al. reported differences of up to 112% between real and simulated performance. It was obvious we could not trust the earlier analysis.

To present more accurate simulated performance numbers, we decided to build a file-system simulator that simulates a file-system in all details, including a disk sub-system back-end much like HP Pantheon disk simulator [31] and Dartmouth's disk simulator [13]. We continuously refined the simulator and eventually we ended up with a full file-system. It turned out that this file system shared lots of data structures and algorithms with our real file system, it only lacked data manipulation code: i.e. the simulator was not an approximation anymore. We repeated the write-saving experiments in this version of the simulator and analyzed the performance numbers again.

We then realized that it is important that a system and its simulator are closely related. When policies and algorithms are different, we cannot be reasonably sure that simulated performance shows real performance. Also, when policies and algorithms are analyzed in a simulator, eventually they will have to be migrated into a real file system. If a real file-system uses completely different data structures or is constructed differently, this migration process usually results in a complete rewrite of the policy or algorithm, which may introduce unwanted side effects or new bottlenecks in a real system. When simulator and file-system are derived from a common framework and use similar, if not equal, data and component structures, migrating code becomes a trivial matter.

We now use Patsy, PFS and the cut-and-paste component library as a new way of doing file system development. Algorithms and file system extensions are first analyzed off-line through pre-recorded file-system traces or hand crafted work loads before they are integrated in a production file system. We learn all the effects of algorithms before they are used.

In fact, we use the framework as a reference system and starting point for further work. Quick off-line file-system experiments are performed before we decide to use the algorithm in a production version of the system. For example, in the framework we are currently performing continuous-media storage experiments, and we plan to use the framework for tertiary storage experiments. We feel that having the framework available gives us a head-start for file-system developments: components are already in-place or are available in the library. Components that are written for experiments are always added to the component library for possible later re-use in other experiments.

Currently, we know of one other system that has integrated a file system simulator and a real system. Thekkath et al. [30] describe a file-system simulator that is able to run an existing kernel file-system in a discrete event simulator. Our work is similar to theirs, but we arrived at the same point through a different route. Thekkath's goal was to lift a kernel file-system into a user level simulator and measure the performance of such a system. We started by building a simulator for some file-system experiments, and later integrated the simulator and a real file-system. In our system, we make file-system *development and measurements* easy, whereas Thekkath's system makes *measurements* of existing file-systems easy. The major advantage of Thekkath's system is that they have measured a production file system (Unix FFS [14]) and calibrated their simulator through a production file-system. We are still using an experimental file-system. A second advantage of Thekkath's system is the availability of a real file-system snapshot before an experiment starts. We are upgrading PFS from an experimental file system to a production file-system and we will use snapshots of PFS in Patsy experiments.

The remainder of this paper is organized as follows. Section 2 describes the common cut-and-paste framework. Section 3 describes the added modules to make up a PFS, and Section 4 describes those components added to make up a file-system simulator. Section 5 we describe a simulator that we built through the frame work, we describe some of the experiments we conducted in this simulator and it describes the lessons we have learned when we were building the system. Section 6 summarizes this paper.

2 Cut-and-Paste

Our goal for the component library is to create a reference file system framework from which we can instantiate many possible file system and simulator configurations. This reference file-system framework, therefore, must be easy to maintain and easy to extend. The cut-and-paste framework should not enforce designers to early algorithms and policy decisions.

To build an *open* component library, we decided to implement the system in an object-oriented language. We have created a set of base components that are used in all file-systems and implement default behavior. Derived components implement specific behavior and are accessed through an inheritance chain. We do not allow derived components to enrich the interface of its base component without adding an interface to its base component. This ensures that interfaces remain clean and without cross dependencies to other than the base component dependencies. By restricting ourselves to such a scheme, our system remains upward compatible.

The cut-and-paste component library is best visualized as a collection of objects that are combined into a real or simulated system whenever they are needed. All components are written in **C**++, are instantiated from their classes and bound to global variables when a system starts. Base components are **C**++ base classes, derived components are **C**++ derived classes. The system currently consists of $\approx$ 55K lines of **C**++ code, divided into 80 classes.

Currently, the following *core* components exist: a *thread scheduler* that allows multiple independent processes in the real or simulated system, a *cache* that implements a full file-system block cache, a *file-system storage* component that defines the storage layout on disk, a *client interface* that defines an abstract front-end to the system, an *abstract file* that "knows" everything about a file when it is loaded into the cache and a *device-driver* that communicates with the simulated or real hardware to read and write data from disk.

For all of the *core* components we have implemented the default behaviour and one or two other algorithms. The core components themselves implement a true *default* file-system. As we are still adding functionality to our system, we expect the number of available algorithms to grow rapidly.

Figure 1 shows most of the core components of the combined file system and simulator. The figure is used as a reference figure: not all components are immediately explained. In particular, all components that are not located inside boxes labeled Patsy or PFS are common to both simulator and system and are part of the cut-and-paste framework. The cut-and-paste components are explained in the remainder of this Section. Boxes labeled PFS are explained in Section 3 and boxes labeled Patsy are explained in Section 4. Since interfaces between objects can be quite large we decided not to describe them here and we refer to the source distribution.

Thread scheduler

The thread scheduler implements threads, synchronization primitives and real or virtual time. Independent file-system processes are given a separate thread of control inside the system. These threads are able

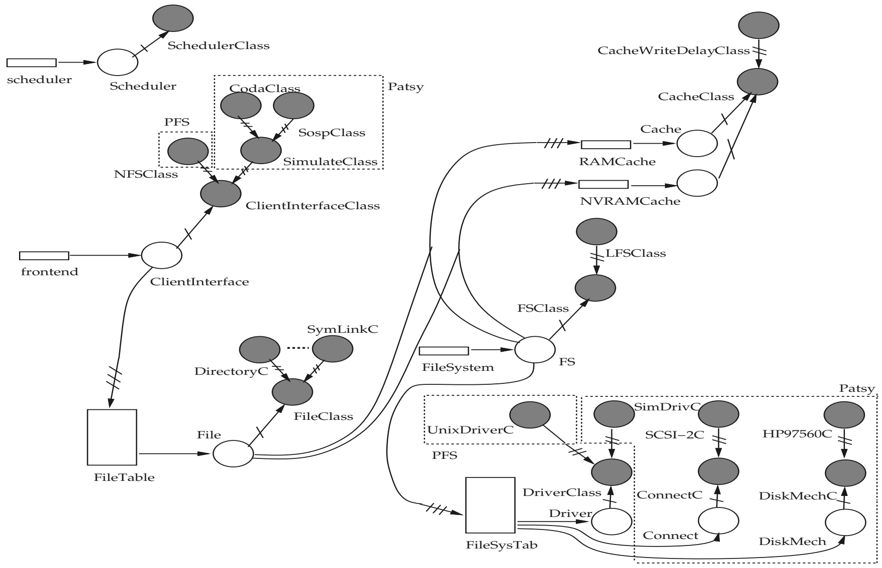

Figure 1: Some of the framework components. Class definitions are shown as shaded ovals, instantiated objects as clear ovals, and global variables and hash tables as clear boxes. Unstriped arrows are plain pointers, single-striped arrows show the relation between the instantiated object and its class definition, double-striped arrows refer to the inheritance chain between the class definitions and triple-striped arrows show which objects call methods in other objects.

to communicate with each other through synchronization primitives: one thread can make others runnable. Threads can also suspend by yielding the processor for some amount of time. Finally, the scheduler knows what the current time is for a real system and it defines virtual time for a simulator.

The synchronization primitives are based on events. Each thread can pick a unique event and block on it. Once a thread has blocked itself, another thread *signals* the event through the scheduler to make the thread runnable again.

Internally, the scheduler maintains a queue of all delayed threads and makes them runnable whenever the timers expire. If the scheduler is configured in a simulator, thread-timers only expire when there are no other threads to run: virtual time is increased to the timer-expire time of the first thread on the delayed queue. When the scheduler is configured in a real system, timers expire in real-time[2].

External events are also managed by the scheduler when it is configured in a real system. In Unix, a file-descriptor is associated with a thread and whenever a message arrives on the file-descriptor, the thread associated with the file-descriptor is made runnable.

Currently, the scheduler provides random scheduling. It picks a random thread from the runnable set, whenever it has to schedule the next thread. When other scheduling policies are required (e.g. when files with real-time constraints are introduced), a derived scheduler class can implement a different scheduling policy.

Caches

The cache modules are used to administer and maintain a file-system block cache. It provides interfaces to administer all dirty, non-dirty and free blocks in lists, and it provides interfaces to allocate blocks from the cache. Also, when blocks are allocated from a full cache, it decides which blocks are replaced and flushed.

The base cache component implements LRU lists to maintain all dirty and non-dirty blocks. Blocks are first allocated from the non-dirty list, and when there are no non-dirty blocks available, the cache initiates a cache flush through the oldest dirty block. To flush, it calls an internal method that may be overloaded by different policies. For example, we have implemented a flush policy that flushes the whole file rather than a single block when a block flush is initiated.

Specific persistency requirements can be implemented in derived components that call into the base component to initiate cache flushes. For example, the Unix SVR4 30-second-update timer policy is implemented through a derived class that examines the contents of the cache every couple of seconds. When

[2]Threads are made runnable in real-time.

it detects that there exists a dirty block older than 30 seconds, it flushes the file associated to the oldest block.

Different cache administration policies are easily implemented by re-implementing the replacement methods of the base-class in a new derived class. For example, to experiment with different replacement policies (e.g. RR, LFU, SLRU, LRU-K or adaptive [12]), only those functions that deal with LRU replacement need to be replaced from the base-class.

The difference between a simulated cache and a real cache is the lack of a data pointer in the simulated case. In all cases where data is moved between buffers, the simulator delays the current thread for the amount of time it would take (based on the system hardware configuration) to copy the data. In a real system, a large chunk of (physical) memory is allocated and divided over all the cache blocks when the system starts.

Storage-layout

The storage-layout component is responsible for defining a file-system layout on a raw disk. This component knows the actual location(s) of file-system metadata, and is able to store and retrieve information from one or more disks. It is consulted whenever something needs to be done with a raw disk.

The base storage-layout class is only an interface: it does not implement an algorithm. Specific layouts are implemented through derived classes. The interface to a storage-layout class is defined such that for all layout and policy decisions, there exists a virtual method in the base-class.

Currently, we have implemented a segmented Log-structured File-System (LFS) [20, 26]. This system stores file-system updates to the end of the log, and is able to find files through an inode-file (IFILE). The log-cleaner can be replaced and is plugged into the LFS component when the system starts up.

To implement other storage-layouts (such as a Unix FFS [14], EFS [28], or journalling file-systems), a new derived storage-layout class needs to be written that defines a new storage-layout on disk. This derived class needs to implement all of the methods defined in the abstract storage-layout class.

A storage-layout module can also be instantiated for a simulator. In this case, all information that would have been read or written to disk is simulated by making educated guesses. If, for example, a file is accessed that is not yet known by the storage-layout module, it picks a random location on disk. Once an initial location has been chosen for a file, the simulator sticks to those addresses.

Abstract Client Interface

The abstract client interface provides the basic file-system interface. There are functions to *open*, *close*, *read*, *write* or *delete* a file and there are functions to manipulate an hierarchical name-space.

The abstract client interface initiates the loading of a file from disk when it is first accessed. It calls into the file system module to read the file's inode into memory. Once the file is in memory, the component stores a reference to it in a global file table.

Furthermore, when the file has been loaded into memory, the abstract client interface maps all incoming user requests to the memory representative of a file.

Files

Abstract client requests are dispatched to so-called *instantiated* files. An instantiated file is used to control a file that has been loaded into the file-system cache. It may contain a memory copy of the file's inode, references to cached file data, and it contains a set of functions to perform operations on a file, such as a *read*, *write* and *flush* method.

As there are many file types in current file-systems (e.g. ordinary Unix-like files, directories, symbolic links, multi-media files, and administrative files) each with different access patterns and behavior, we have implemented each file type in a separate derived class. All components derive basic file functionality from the base file. This structure allows us to separate policies that optimize access to a particular file type. An example of this is a multi-media file. If ordinary cache policies are used on a multi-media file the whole cache would fill up with this data. A multi-media file prevents this from happening by implementing other cache policies.

Pei Cao *et al.* has shown that two-level file-caching can be beneficial to overall file-system performance [7]. In this work, cache replacement policies are delegated to a user-level manager process. In our approach we have the opportunity to delegate the policy decisions to a particular file – the manager can be implemented inside the file-system. When a file is opened, a client can inform the file-system to use a certain replacement policy. This approach can give us a fine-grain control over cache replacement policies.

When a file is requested by a client, the file-system front-end examines the file type of the requested file (through its inode or other administrative means) and instantiates an object of that type to manage the file while it is in core.

A file is called *active* if the instantiated file spawns a thread of control that works independently inside the file-system. Such functionality is especially useful if files are manipulated that have timing constraints on them. In a multi-media file, for example, the thread of control may take care of cache pre-loading and can even negotiate with (remote) clients and other modules in the file-system to establish a certain *Quality of Service (QoS)*.

3 Pegasus File-system

The base components in the cut-and-paste library do not make up a complete system: they lack interfaces to the environment. To complete such a system, *helper* components are added to the component library that glue all the components into the environment.

The system glue currently only consists of two parts: the system needs a *real* user interface, a PFS client interface and it requires a real disk-driver to access a real disk.

PFS Client Interface

We use NFS [23] as the external PFS interface. We have constructed a full NFS client interface class, which is a derived class from the abstract client interface class. The NFS class spawns a number of threads that wait for incoming mount and NFS requests. Whenever a request is received, the call is dispatched to one (or more) calls in the abstract client interface. Each thread in the NFS component acts as a representative of a client while the request is in progress.

In the future we will construct a client interface that enables strict consistency. This client interface will use many of the techniques that are already in use by Sprite [16] and Mike's File System (MFS) [6]. By using client caching we hope to reduce the amount of network traffic and file latency. We will realize this within the cut-and-paste framework so that we can simulate client/server interaction and client cache performance.

Disk-driver

Real disks are accessed through disk-drivers. Disk-drivers implement one or more disk queues and send new operations to disks whenever they are ready to service new requests. They can implement disk queue scheduling policies to optimize disk I/O queue time (e.g. SCAN, C-SCAN, LOOK, C-LOOK [11, 25]) or guarantee real-time delivery of data through algorithms such as scan-EDF [18].

Currently, only one disk-driver exists. This driver implements a combined read-write queue and schedules I/O requests through the C-LOOK scheduling policy. It uses a Unix-file (ordinary file, or raw-device) as back-end.

4 Patsy

Patsy is the instantiation of the cut-and-paste library to a file-system simulator combined with some helper components for off-line file-system simulation. In particular, we added components that simulate real disk-drivers and disks, a component that simulates the connection between the host and disk sub-system, and a component hierarchy that is able to read a particular trace file and dispatch it to the simulator. We are simulating only a small subset all hardware types. We hope to integrate our file-system simulator into HP's Pantheon disk-simulator and use that as our disk back-end because Pantheon simulates a wide variety of storage hardware.

Simulated disk-drivers

Simulated disks are accessed through simulation disk-drivers. These disk-drivers provide the same functions as their real counterparts, but also provide mechanisms to simulate the sending and receiving of operations from disk. The simulated disk-drivers have exactly the same interface as a real disk-driver: the differences are in the internal implementation. The system itself does not know it is communicating with a "fake" disk.

Simulation disk drivers package disk operations in I/O-request data structures. The I/O-request data structures contain all the relevant information for the disk simulator to simulate a disk read or write and contain timing information to measure the performance of the I/O operation.

Before an operation is activated on disk, the disk-driver acquires the host/disk connection and simulates the sending of data. This is required as many more entities can make use of the same host/disk connection. If the connection is already in use, the disk driver waits until the connection is released again. Finally, it sends the I/O request to the disk itself and activates it.

The disk will perform the I/O request and later transmit the results to the disk-driver. For this, the disk acquires the host/disk connection and simulates the transmission of the I/O request back to the driver. Finally, the simulated disk-driver resumes the original caller and informs it of the I/O results through the I/O request data structure.

Simulated disks

The disk component in the simulator acts as a representative for a real disk. A simulated disk component knows about heads, tracks, sectors, rotational speed, controller overhead and it may implement disk cache policies.

Internally, a disk is modeled by a separate thread of control that waits for work to arrive from external sources. Whenever a read or write request arrives at the disk, the controller unpacks the request, seeks to the correct cylinder or switches heads. Next, the disk waits for the rotational delay and reads or writes data to disk. Finally, the disk transfers the data to the host, or signals the host that a write has completed.

It is obvious that in the simulated world, no real data is moved to and from disk. We simulate this by delaying the thread of control by the amount of time it would have taken to transfer the data.

Currently, we have implemented an HP97560 disk [21, 13] as a separate disk class. This disk is equipped with a 128KB internal cache that can be used for immediate reported writes (writes that complete once data arrive in the disk's internal cache) and a read-ahead policy (when there are no more outstanding requests, the disk reads the next 4KB following the last read).

Connections

Connections are the links between the host and the disk sub-system. Connections are used to transfer data, requests and responses between disk and hosts. They also arbitrate if there is more than one controller that wants to send data over the same connection to simulate connection contention (e.g. SCSI bus contention).

Again, as no real data can be moved through a connection, the connection delays the thread of control by the amount of time it would have taken to transfer the data through the connection.

We have implemented a SCSI-2 bus [24]. This bus allows multiple hosts/disks to use the same connection, and it allows hosts/disks to disconnect and reconnect during a single SCSI transaction. The bus simulates a bus transfer speed of 10MB/s.

Work loads and traces

File-system traces are collections of records that describe all the activity of a real file-system at some time. These records specify when the operation took place (usually down to the microsecond), and which file-system operation was executed. Usually, file-system traces do not present all the details that are required for an exact replay. The reason is that, although all parameters can be recorded, doing so would result in prohibitively large trace files: the recording of such trace files would influence ordinary file-system performance too much [15, 17]. When replaying traces, we synthesize those parameters that are missing as best we can (e.g. the initial location of a file on disk, file names, initial layout of the file-system, the exact time a read or write was executed).

We are also considering a component that can be used to hand craft work loads using probabilistic means. This component will, given some inputs, generate a work load and dispatch it to the simulator. The advantage of such a component is that it will improve our confidence in the simulation results.

We have modeled a trace simulation class hierarchy on top of the abstract-client interface. All file-system requests recorded in the file-system traces are mapped to calls in the abstract-client interface. The simulator components currently consist of three parts: a general simulation class that provides performance results gathering and dispatches operations to the abstract-client interface. Furthermore, there are two derived classes: a Sprite class to replay the Sprite traces [2] and a Coda class to replay the Coda traces [15].

Clients are modeled by separate threads of control inside the Sprite and Coda classes. The threads read a part of the trace file, group operations that obviously belong together (such as an *open, read, read, write, ..., close* sequence), and call the abstract-client interface to execute the operation on the simulated system. Since all of the trace records have timing information in them, the threads know how long they have to delay themselves before they can dispatch the next operation.

When simulation information is missing (such as the actual time a *read* or *write* operation took place), the client thread makes a guess. In the case of the missing *read* and *write* times, the operations are positioned equidistant between the *open* and *close* operation. We plan to experiment with the position of these operations.

The overall measurements are taken from the general simulation class. This class measures how long it takes before an operation completes. The measurements are shown every 15 minutes of simulation time and of the overall simulation.

Detailed internal measurements are provided by plug-in statistics objects. These plug-in statistics can be activated when the simulator is started and they can provide standard statistics output with or without histograms. Some of the standard detailed statistics objects include histograms of disk queue sizes, cache statistics, and disk rotational delay statistics.

5 Using the component library

We have used the component library for a file-system performance experiments and to construct a real file-system. We built the algorithms for the various experiments in the simulator, and later we instantiated the same algorithm for a real system: we did not have to change anything in the code except for some small additions when data was actually moved.

In this section we describe an experiment we conducted in an instantiated simulator and some of the lessons we learned when we built the component library. The experiments primarily show what kind of simulations are supported. The full results that are presented briefly in this section are subject of another report. We only present the final results of the off-line experiments.

5.1 Delayed write policies

One of the reasons to build a file-system simulator in the first place was to re-do the performance analysis of various delayed write policies, based on ideas described in earlier work [4]. Also, we have conducted the experiments to confirm cache experiment results as reported by Ousterhout [17]. In those experiments, we buffered dirty data longer in the cache with the hope that less data is written to disk. If dirty data stays longer in the cache without being written to disk, changes are higher that file *delete* and *truncate* calls remove the file's dirty data before a flush policy gets a change to write the data to disk. If less data is written to disk, disk queues become shorter and I/O latencies decrease.

If the fraction of dirty data in the file-system cache increases, read cache hit-rates may be negatively influenced. If there is more dirty data in the cache, non-dirty data is replaced earlier. If overall cache hit-rates drop (i.e. if cache hit-rates are higher for non-dirty caches), more data is read from disk. This leads to longer disk queues and to increased I/O latencies. We analyze this trade-off.

If dirty data remains longer in the cache before being written to disk, more data is lost when the system fails. Earlier work describes how to protect dirty data from a single point of failure in the system through client write caching and by using non-volatile RAM (NVRAM) or a uninterruptable power supply (UPS) in the file-server [4].

We are performing four different experiments with the Sprite traces to analyze the performance effects of these *write-saving* policies. Our first experiment shows the base line performance of an ordinary Unix file-system with a 30-second-update policy. Dirty data is buffered 30 seconds before it is sent to disk (the write-delay experiment). Next, we equip the file-system with a UPS and only flush a cache block when we are out of non-dirty cache-blocks. This experiment shows the other extreme as blocks are only written when the memory is filled with dirty blocks. Finally, we equip the file-system with 4 MBs of NVRAM and we disallow dirty data to reside in volatile-RAM. If the NVRAM is full and we need free (or non-dirty) blocks in the NVRAM buffer, we flush the oldest dirty block to disk. For the NVRAM case we consider two flush policies: we either flush the whole file associated with the oldest block (with the expectation that on average there is more non-dirty data in the cache), and we flush only the oldest block.

We have included the NVRAM experiment to answer how much file-system latencies improve by using a small NVRAM buffer in the file-server. This question was raised in Baker et al. [1] and left unanswered.

For all experiments, we expected the UPS experiment to perform best. The reason for this is that all of the available cache can be used for write-caching, minimizing the amount of written data. For the other experiments we did not know on beforehand which policy would work best: if too much dirty data is generated, the NVRAM buffer may be the system's bottleneck and reducing the write-delay time when the file-system is busy. In fact, adding NVRAM to the file-system may be counter productive.

We have rebuilt the Sprite file-server as closely as we could [29, 9] through the cut-and-paste component library and we ran the recorded Sprite traces on this version of Patsy. The original machine on which the traces were recorded was a Sun 4/280 equipped with 128MB of main memory, and three SCSI busses that connect to a total of 10 disks. There were a total of 14 file-systems on the set of disks, of which two were clearly hot-spots. For the experiments we have used simulated HP97560 disks and SCSI-2 busses. On all file-systems we ran a segmented LFS. All components used by this simulator are derived from the cut-and-paste framework.

Figures 2–4 show a cumulative distribution of the file-system latencies for the Sprite traces in Patsy (only traces 1a, 1b, and 5 are shown, the others are permutations of these three runs). Each trace represents a 24-hour period of operations and we have measured the time it takes to complete each of the operations. The figures show a cumulative distribution of the latencies. The horizontal bars show the average experienced latency, the vertical bar shows the fraction of operations completed. Figure 5 shows all mean file-system latencies for all the traces.

Each graph in Figures 2–4 have a similar form. All operations that complete within 2-milliseconds

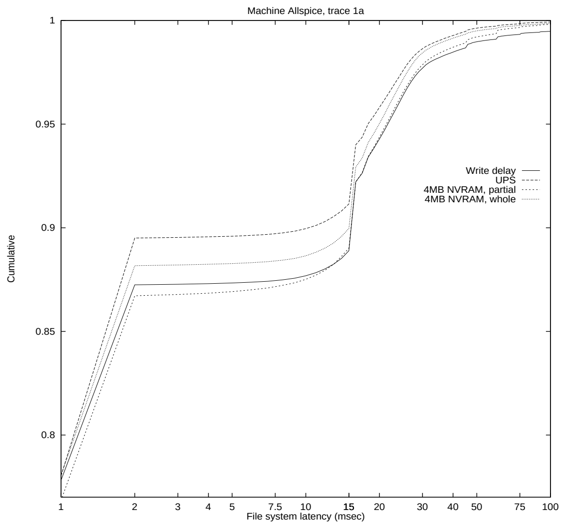

Figure 2: File-system latencies, trace 1a

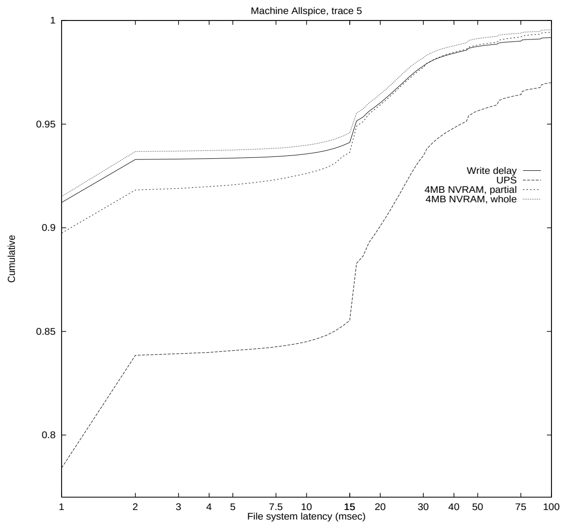

Figure 4: File-system latencies, trace 5

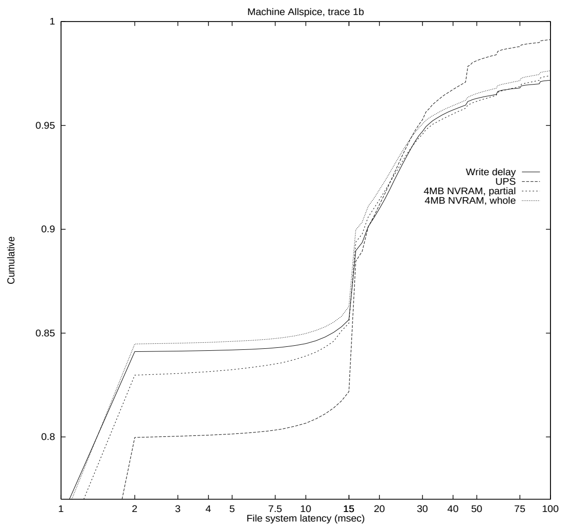

Figure 3: File-system latencies, trace 1b

are serviced from the file-system caches. The 2-milliseconds boundary is the minimal latency when a request is serviced by the disk (SCSI-request decoding). The period up to 17-milliseconds represents the time waiting for the rotation on disk (HP97560 disks spin at 4002 rpm), including the controller overhead. The bump at 17-milliseconds represents the large amount of operations that had to wait for a full disk-rotation. The periods larger than 17-milliseconds are those when the disk queues were longer than one entry or when the disk required head and/or cylinder switches. We are re-evaluating the trace results to present both delays separately.

From the performance numbers shown in Figures 2–4 and Figure 5 we learn that, in general, the UPS experiment performs better than the NVRAM experiments, which in turn performs better than the ordinary write delay policy. In all but trace 5, a UPS file-system is much faster than the write delay experiment while the NVRAM experiment is only twice as fast. The reason for this is that for most traces disk queues are minimized.

During trace 5, many large *writes* enter the system while there are also a fair amount of *stat* and *read* operations. The write operations fill up all of the available memory (which are not flushed since we are using a naive flush policy) and clutter up memory. This lowers read cache hit rates[3], and cause the read operations to stall while data is flushed to disk. This happens to a lesser extent in trace 1b. We found that in general write-saving policies combined with a naive flush policy resulted in lower cache hit rates.

Figure 5 shows that for Trace 1b NVRAM does not help much to reduce file system latencies. In this trace there are many large and parallel write operations. Since dirty data can only be stored in NVRAM, the NVRAM becomes a bottleneck: new writes are waiting for the NVRAM to drain. In some cases we found that the write-back policy deteriorated to a write-through policy.

For the NVRAM case, we measured two different flush policies: whole file (that is: when a file-block is flushed, all dirty blocks of that file are flushed), and partial file (only the selected block is flushed). As expected, whole-file flushes perform better than partial-file flushes as it leaves on average more non-dirty space in the NVRAM buffer. This means that write operations complete sooner as they do not have

[3]Not shown.

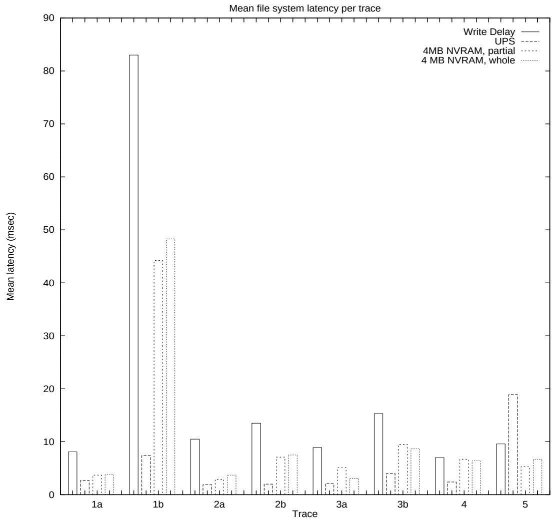

Figure 5: Mean file-system latencies

to wait for data to be flushed to disk first.

In general, delaying write operations reduces disk contention even though there are more cache misses. We showed that because of this, file-system operation latencies decrease. When there is lots of activity in the file-system, there is a possibility that the available memory is cluttered up with dirty data, which increases sub-sequent file-system latencies as data needs to be written to disk first. To solve this problem, we are experimenting with more aggressive write policies.

5.2 Lessons learned

While building and experimenting with an off-line simulator, we were surprised by a number of performance bottlenecks that would have been hard to find in an on-line file-system. Since simulator and system are one and the same, solving the performance bottleneck in the simulator, also solves the performance bottleneck in the real system.

We ran the Sprite traces on the simulator and we were surprised by the "slowness" of the simulator: it took several hours to simulate one hour of Sprite time. First, we used *prof(1)* to analyze which procedures were accessed most. It turned out that the way we were maintaining the LRU lists was sub-optimal. By carefully analyzing what the system was doing through *gdb(1)*, we detected several short-cuts in list maintenance. This improved simulation time dramatically. To look for this problem in a real system would have been hard. First, it would have been hard to continuously replay the same workload. Second, it would have been hard to run the real system under *gdb(1)* and to analyze step-by-step what the actual problems are.

A second problem we found had to do with the way a cache was flushed. In our original system, the thread that needed a cache block was also the one that initiated a cache flush and waited for the flush to complete. As more esoteric flush policies were used, the delay for this thread increased. Since we were able to analyze off-line we were able to quickly analyze why some threads were severely delayed. The obvious solution was to make the flush policy an a-synchronous operation. Again, this problem would have been more difficult to find in a real system.

We found the NVRAM contention problem through carefully analyzing and hand-crafting a work load. The other option would have been to allocate a part of a file-server's memory and simulate the behavior of NVRAM in an on-line system. By being able to simulate behavior off-line, we were able to quickly decide that it is better to equip a file-system with a UPS rather than NVRAM.

The biggest surprise was when we instantiated the component library to a real system: most algorithms ran immediately. This basically means that file-systems can be developed in a controlled environment. The file-system is now is use for real data storage.

It is certainly true that file-system experiments can also be performed in kernel-based file-systems. Current BSD systems can easily be changed to support all kinds of experiments (as noted by an anonymous reviewer). We believe, however, that user-level file-systems are easier to maintain and it is easier to track performance bottlenecks. Developing file-systems in user mode has the advantage that the machine does not halt whenever there is a fatal error. A production version of the system can always be implemented inside the kernel, although we do not expect the performance to improve much. The problem with Unix user-level file-systems, however, is that some parts of the file-system are constructed carefully to hide the absence of threads in most Unix systems. To demonstrate that the current system can also be used as a kernel service, we have inserted the system into the Nemesis micro-kernel [19]. The reason for this integration is to provide Nemesis with a file-system and to perform multi-media storage experiments.

We fully agree that the system described in this paper is a partially an engineering practice (as noted by an anonymous reviewer): we have built a tool to perform file-system experiments. Since related parts of the system are encapsulated in their class hierarchy, changing particular file-system policies is not hard. This gives us an opportunity to quickly learn the effects of new algorithms and file-system configurations (even if we do not have the equipment we are simulating).

5.3 How to make the results usable

Based on the experiments, we have decided to pursue a real file-system that is equipped with a UPS. We are re-evaluating a protocol to safely distribute dirty data over client and server machine. As is shown in earlier work [4], this allows us to guarantee data persistency in case of a single point of failure, or a global power failure. Evidently, we will re-evaluate this through the simulator.

Once we are happy with the performance results of a more aggressive flush policy, we will migrate the cache policy into PFS and measure the performance relative to a standard 30-second-update policy. In these measurements we will show the performance differences between a simulator and a real system.

We have compared performance differences of system and simulator in a small test environment. The analysis so-far suggests that the results in the simulator have real value and are comparable to real performance. We are working on a full comparison of a simulator and system and we will provide a separate report that validates our approach.

6 Summary and further work

We have presented a file-system reference framework, which we use to experiment with new file-system storage algorithms. The file-system framework can be instantiated to off-line simulators and on-line file-systems. Through off-line simulations, storage algorithms are analyzed thoroughly before being used in a real system. We have constructed the framework such that it is reasonably straightforward to analyze other researcher's algorithms in our simulator.

As an example, we have analyzed several write-saving policies in a simulator, that is instantiated from the framework. We concluded that write-saving policies are beneficial even though cache-hit rates decrease. Currently, we are still analyzing more aggressive write policies before we will put the policies to use in a real system and measure the relative performance of a real system compared to an equivalent simulator.

Acknowledgments

We would like to thank the anonymous reviewers, Tage Stabell-Kulø, Richard Golding, Sjoerd Mullender, Arne Helme, George Neville-Neil, Paul Sijben, Martijn van der Valk, Lars Vognild and the other Pegasi for their support and criticism on earlier versions of this paper.

Availability

The framework is available through the authors. Announcements are made on (**pfs[-request]@ pegasus.esprit.ec.org**).

References

[1] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of Fifth ASPLOS* (Boston, Massachusetts), pages 10–22, 1992.

[2] 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.

[3] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Conference Proceedings of of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 277–88. Usenix Association, 16–20 January 1995.

[4] 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.

[5] Peter Bosch. From ULFS towards PFS. University of Twente, 9 December 1992. A list of notes used to design a new file system.

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

[7] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. *Proceedings of First Symposium on Operating Systems Design and Impl. (OSDI)* (Monterey, CA), pages 165–77. Usenix Association, 14–17 November 1994.

[8] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Conference Proceedings of of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 201–12. Usenix Association, 16–20 January 1995.

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

[10] 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.

[11] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical report HPL–CSP–91–7rev1. Hewlett-Packard Company, 16 February 1991.

[12] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance. *IEEE Computer*, pages 38–46, March 1994.

[13] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical report PCS–TR94–220. Dartmouth College, 18 July 1994.

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

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

[16] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, **6**(1):134–54, February 1988.

[17] 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 the 10th ACM Symposium on Operating Systems Principles* (Orcas Island WA (USA)). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[18] A. L. Narasimha Reddy and James C. Wyllie. I/O Issues in a Multimedia System. *IEEE Computer*, pages 69–74, March 1994.

[19] Timothy Roscoe. The Structure of a Multi-Service Operating System. Technical report 94–5. Pegasus project, Esprit BRA 6586, March 1994.

[20] 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* (Pacific Grove, CA, USA, October 1991). Published as *SIGOPS*, **25**(5):1–15. Association for Computing Machinery, October 1991.

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

[22] Chris Ruemmler and John Wilkes. UNIX Disk Access Patterns. *USENIX Technical Conference Proceedings* (San Diego, CA), pages 405–20. USENIX, Winter 1993.

[23] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. *USENIX Conference Proceedings* (Portland, OR), pages 119–30. USENIX, Summer 1985.

[24] Small Computer System Interface – 2. American National Standard for Information systems, 23 June 1989. Working draft proposal.

[25] Margo Seltzer. A comparison of data manager and file system supported transaction processing. Computer Science Div., Department of Electrical Engineering and Computer Science, University of California at Berkeley, January 1990. Obtained from the author.

[26] 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.

[27] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Conference Proceedings of of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 249–64. Usenix Association, 16–20 January 1995.

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

[29] Ken Shirriff. Allspice's configuration, 23 March 1995. Private communication.

[30] Chandramohan A. Thekkath, John Wilkes, and Edward D. Lazowska. Techniques for file system simulation. *Software—Practice and Experience*, **24**(11):981–99. John Wiley and Sons Ltd, November 1994.

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

Author information

Peter Bosch graduated with degrees in EE (Bc., 1988), and CS (M.Sc., 1994) from the Univ. of Twente, NL. He has worked since 1991 for the Univ. of Twente as a Ph.D. student. His current work involves research into file-systems and the implementation of a high-performance file-system for the ESPRIT PEGASUS project. His hobbies include traveling and spending evenings in restaurants.

Prof. Sape J. Mullender is chairman of the systems programming and architecture department in the Faculty of Computer Science of the University of Twente in the Netherlands, where he leads the Huygens research project on fault-tolerance, real time, multimedia and security in distributed systems.

Sape Mullender is interested in high-performance distributed computing and the design of scalable fault-tolerant services. He is also concerned about organization and protection in distributed systems that can span a continent. He is a principal designer of the Amoeba distributed operating system.