KFS: Exploring Flexibility in File System Design

Dilma M. da Silva[1] **, Livio B. Soares**[2*]**, Orran Krieger**[1]

[1]IBM TJ Watson Research Center
PO Box 218
Yorktown Heights, NY 10598
USA

[2]Computer Science Department
University of São Paulo
Rua do Matão, 1010
São Paulo, SP CEP 05508-900
Brazil

dilma@watson.ibm.com, livio@ime.usp.br, okrieg@us.ibm.com

Abstract. *This paper argues that file systems need to be customizable at the granularity of files and directories in order to meet the requirements and usage access patterns of various workloads. We present KFS, a file system designed to allow for fine-grained adaptability of services. In KFS, each file or directory may have its own tailored service implementation, and these implementations may be replaced on the fly. KFS runs under Linux and the K42 research operating system. We describe how KFS's flexibility has been explored to achieve better schemes for meta-data management and consistency.*

Resumo. *Este artigo discute a necessidade de sistemas de arquivos que sejam customizáveis no nível de arquivos e diretórios, de forma a atender aos requisitos e padrões de acessos de várias cargas de execução. Nós apresentamos o KFS, um sistema de arquivos projetado de forma a possibilitar adaptabilidade de serviços com granularidade fina. Em KFS, cada arquivo ou diretório pode ter sua própria implementação de serviço, feita sob medida para suas necessidades. Estas implementações podem ser substituídas dinamicamente. KFS está disponível para o sistema operacional Linux e o sistema operacional de pesquisa K42. Nós descrevemos como a flexibilidade do KFS pode ser explorada de forma a obter esquemas melhores para gerenciamento de meta-dados e consistência.*

1. Introduction

Traditionally, file systems are designed to address a specific set of requirements and assumptions about file characteristics, expected workload, usage and failure patterns. The choices for persistent data representation and manipulation policies usually apply uniformly to all elements in the file system, regardless of particular characteristics. The file system implementation has to handle correctly all expected corner cases, and in the traditional design (e.g., FFS [McKusick et al., 1984], ext2fs [Card et al., 1994]) often the price for such generality is a higher performance cost for the most common scenarios, for example:

- studies have shown that for many workloads, the majority of files are small [Ousterhout et al., 1985, Mesnier et al., 2004]. By optimizing file manipulation for small sizes, an improvement of 40% has been reported in [Soules et al., 2003];

*Partially supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), Brazil, through grant 132845.

- The NFS traces collected in [Mesnier et al., 2004] indicate that more than 63% of the files have a lifespan of less than 1 second; storing such files in non-volatile RAM (NVRAM) would result in considerable performance improvement;
- concurrent write access for the same file is not common (existing traces indicate less than 1%), but general file system paths include acquisition and release of locks.

Also, file systems have been offering replication, striping over a set of disks or file versioning as a policy compassing the whole file system, although in many scenarios the owner of the data may care about such requirements for only a part of her data.

In order to address these issues in traditional file systems, we propose that fine-grained flexibility be the stem of the architecture for a file system. A fine-grained flexible file system would allow implementations that optimize for very specific scenarios, access patterns, and requirements. It would also allow different, and potentially conflicting, implementations and policies to coexist in the same file system.

Flexibility is also desirable to address requirements that are not known yet, e.g., to extend the file system to handle new file access patterns that are introduced by new classes of applications.

In our research, we are investigating a file system framework based on a very flexible design with an implementation that is "real" enough to run scientific and commercial workloads, achieving performance comparable to other file systems such as Linux's ext2fs. Our main research goal is to investigate dynamic customization of file system services. Also, we want the design to be flexible enough to allow us to experiment with new storage technology. For example, new technology as MEMS-based storage devices [Schlosser and Ganger, 2004] may change considerably our current assumptions on disk capacity, latency, and bandwidth. Also, new "smart devices" may be able to carry out some of the management currently implemented in the file system layer. For example, current object-based storage [Anderson, 1999] design moves block allocation and request scheduling responsibilities from the file system layer into the device layer.

Previous work has addressed the issue of building file systems to address new requirements, focusing on the ease of development. Maziéres describes in [Mazieres, 2001] a toolkit for extending Unix file systems. This work exposes the NFS interface, allowing new file systems to be implemented portably at user level. Stackable file systems promise to speed file system development by providing an extensible file system interface, allowing new features to be added incrementally [Rosenthal, 1990, Heidemann and Popek, 1994, Khalid and Nelson, 1993, Zadok and Nieh, 2000]. All these efforts focus on evolving behavior for the whole file system; our research addresses changing behavior in the granularity of files or directories.

Our research builds on the research done by the Hurricane File System (HFS) [Krieger and Stumm, 1997]. HFS was designed for (potentially large-scale) shared-memory multiprocessors, based on the principle that, in order to maximize performance for applications with diverse requirements, a file system must support a wide variety of file structures, file system policies, and I/O interfaces. As an extreme example, HFS allows a file's structure to be optimized for concurrent random-access write-only operations by 10 threads, something no other file system can do. HFS explored its flexibility to achieve better performance and scalability. It proved that its flexibility comes with little processing or I/O overhead.

This paper presents the K42 File System (KFS). KFS has been strongly influenced by HFS. In KFS parallel performance and scalability are important, but we also explore flexibility in terms of other requirements such as replication of data, meta-data consistency, and dynamic changes in file representation to match detected file access patterns.

KFS is part of the K42 project [k42, 2004]. K42 is an open-source research OS for cache-coherent 64-bit multiprocessor systems. It uses an object-oriented design to achieve good performance, scalability, customizability, and maintainability. K42 supports the Linux API and ABI [Appavoo et al., 2003a] allowing it to use unmodified Linux applications and libraries. The system is fully functional for 64-bit applications, and can run codes ranging from scientific applications to complex benchmarks like SDET to significant subsystems like Apache. The support for 32-bit applications is close to completion.

KFS is available for both the K42 and Linux [linux, 2004] operating systems. The source code for KFS is available as part [1] of K42 at `http://www.research.ibm.com/K42/k42cvs.html`.

In this paper we present KFS, and describe two initiatives in exploring its flexibility. Section 2 provides an overview of KFS. Section 3 presents a new scheme for meta-data consistency implemented in KFS, and shows performance data contrasting KFS with other file systems. Section 4 discusses our work on exploring KFS flexibility to achieve decentralization of meta-data information, so that data and meta-data can be collocated. Section 5 concludes by describing other on-going research initiatives on KFS.

2. Overview of KFS

KFS, as other components of K42, uses a building-block approach [Auslander et al., 1997], which permits a high degree of customization for applications.

In KFS, each virtual/physical resource instance (e.g., a particular file, open file instance, block allocation map) is implemented by combining a set of (C++) objects. For example, for KFS running on K42 there is no global page cache or buffer cache; instead, for each file, there is an independent object that caches the blocks of that file. When running on Linux, KFS is integrated with Linux's page and buffer cache. The object-oriented nature of KFS makes it a particularly good platform for exploring fine-grained customization. There are no imposition of global policies or data structures.

Several service implementations (with alternative data structuring and policies) are available in KFS. Each element (e.g., file, directory, open file instance) in the system can be serviced by the implementation that best fits its requirements; if the requirements change, the component representing the element in KFS can be replaced accordingly[2]. Applications can achieve better performance by using the services that match their access patterns and synchronization requirements.

When a KFS file system is mounted, the blocks on disk corresponding to the superblock are read, and a *SuperBlock* object is instantiated to represent it. A *BlockMap* object is also instantiated to represent block allocation information. Another important object instantiated at file system creation time is the *RecordMap*, that keeps the association between file system elements (*inodes* is the usual terminology in Unix file systems) and their disk block allocation. In traditional Unix file systems, the inode location is fixed: given an inode number, it is known where to go on the disk to retrieve/update its persistent representation. In KFS, the data representation format for a given inode may vary during its lifetime, potentially requiring a new location on disk. In KFS, as in the Log-Structured File system (LFS) [Rosenblum and Ousterhout, 1992], inode location is not fixed. The *RecordMap* object maintains the inode location mapping. It also keeps information about the type of implementation being used to represent the element, so that when the file/directory is retrieved from disk the appropriate object is instantiated to represent it.

[1] KFS is in directory os/servers/kfs of K42's source code tree.

[2] In K42, actively used object implementations can be replace on the fly[Appavoo et al., 2003b].

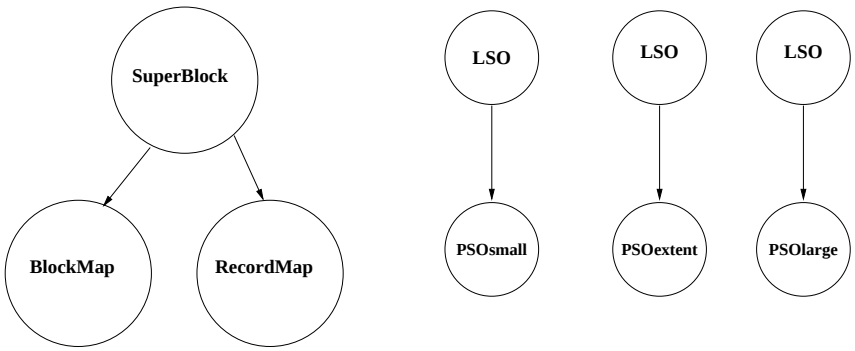

Figure 1: Objects representing files with different size and access pattern characteristics in KFS.

An element (file or directory) in KFS is represented in memory by two objects: one providing a logical view of the object (called Logical Server Object, or LSO), and one encapsulating its persistent characteristics (Physical Server Object, or PSO).

Figure 1 portrays the scenario where three files are instantiated: a small file, a very large file, and a file where extended allocation is being used. These files are represented by a common logical object (LSO) and by PSO objects tailored for their specific characteristics: PSOsmall, PSOextent, PSOlarge. If the small file grows, the PSOsmall is replaced by the appropriate object (e.g., PSOlarge). The *RecordMap* object is updated in order to reflect the new object type and the (potential) new file location on disk.

KFS file systems may spawn multiple disks. Figure 2 pictures a scenario where file X is being replicated on the two available disks, while file Y is being striped on the two disks in a round-robin fashion, and file Z is also being replicated, but with its content being compressed before going to disk.

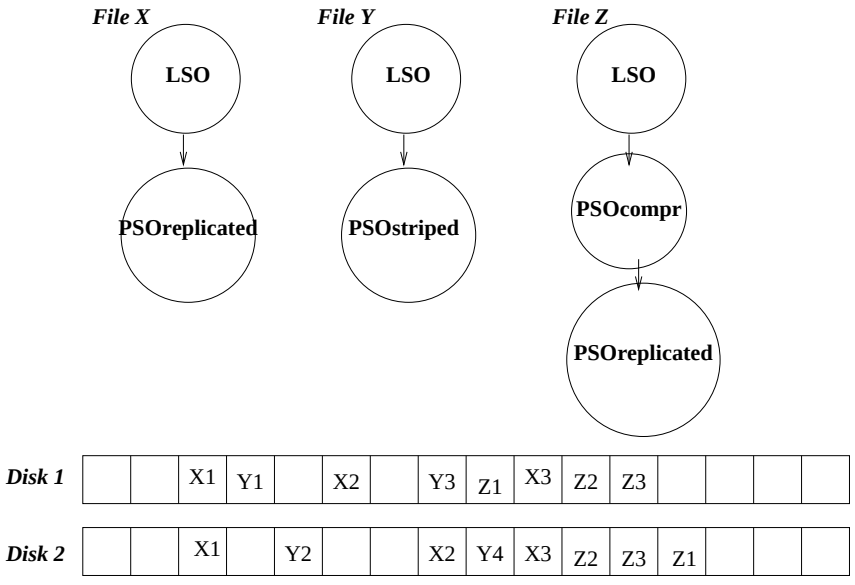

Figure 2: KFS objects and block allocation for files X (replicated; blocks X1, X2, X3), Y (striped; blocks Y1, Y2, Y3, Y4), and Z (compressed and replicated; blocks Z1, Z2, Z3).

In the current KFS implementation, when a file is created the choice of object implementation to be used is explicitly made by the file system code based on simplistic heuristics. Also, the user could specify intended behavior by changing values on the appropriate objects residing on /proc. As many researchers have pointed out, the POSIX API lacks the capability of associating arbitrary attributed to files; such attributes would be a better way for the user

to provide hints about the data elements they are creating or manipulating. Libraries such as MPI-IO [Corbett et al., 2002] already provide a hook for passing out such hints.

3. Exploring flexibility: a new meta-data consistency scheme

File system consistency, in the presence of system crashes, has always been a strong concern in the operating system community. Availability, integrity, and performance are commonly the main requirements associated with file system consistency. Consistency in file systems is achieved by performing changes to the on-disk version of meta-data in a consistent manner. That is, when a system crash occurs, the on-disk version of the meta-data should contain enough information to permit the production of a coherent state of the file system.

Several novel software-based approaches for solving the meta-data update problem have been studied and implemented. One of these approaches in particular, journaling [Hagmann, 1987], has been implemented and is in use in a wide range of server platforms today. Other approaches, such as log-structured file systems [Rosenblum and Ousterhout, 1992] and soft updates [Ganger and Patt, 1994] have had less adoption by commercial operating systems. The soft updates technique has been incorporated into the 4.4BSD fast file system [McKusick et al., 1984], and we have no knowledge of the incorporation of log-structured file systems on any commercial operating system. The implementation complexity imposed by these mechanisms is a disadvantage that may inhibit their adoption and inclusion in commercial file systems. Even though there is evidence [Seltzer et al., 2000] that the soft updates approach can achieve better performance and has stronger integrity guarantees than journaling file systems, the industry so far has adopted the latter.

We proposed *meta-data snapshotting* as a low-cost, scalable, and simple mechanism that provides file system integrity. It allows the safe use of write-back caching by making successive snapshots of the meta-data using copy-on-write, and atomically committing the snapshot to stable storage without interrupting file system availability. In the presence of system failures, no file system checker or any other operation is necessary to mount the file system, therefore it greatly improves system availability. The flexibility of KFS's design and its separation of concerns simplified the task of adding such new protocol into KFS. We showed that meta-data snapshotting has low overhead: for a micro-benchmark, and two macro-benchmarks, the measured overhead is of at most 4%, when compared to a completely asynchronous file system, with no consistency guarantees [Soares et al., 2003]. Our meta-data snapshotting algorithm also induces less overhead then a write-ahead journaling file system, and it scales much better when the number of clients and file system operations grows, as shown in Figure 3.

Figure 3 portrays the behavior of the *Dbench* benchmark, an open-source benchmark, very much used in the Linux development community. Its creation was inspired by the Netbench [Swartz, 1997] benchmark. Netbench is a bench created to measure the performance of Windows file servers implementations, such as WindowsNT [Custer, 1994] and Samba [Blair, 1998]. The problem with this benchmark is that it requires a large number of clients to produce a significant load on the tested server. Dbench, therefore, is a synthetic benchmark which tries to emulate NetBench, and the load imposed of such a file server. It involves file creation, deletion, and renaming; reading, writing and flushing out data; directory creation and removal; and retrieval of information about files and directories (through the stat system call). The dbench benchmark allows one to specify the number of concurrent clients to run.

Figure 3 provides performance data for dbench running on four file systems (all running on top of Linux 2.4.22): *ext2fs* [Card et al., 1994], *ext3fs* [Tweedie, 1998], KFS (without any consistency guarantees, like ext2), and KFS+snap, the version of KFS with meta-data snapshotting capability. *Ext2fs* was chosen because, besides being the default file system in

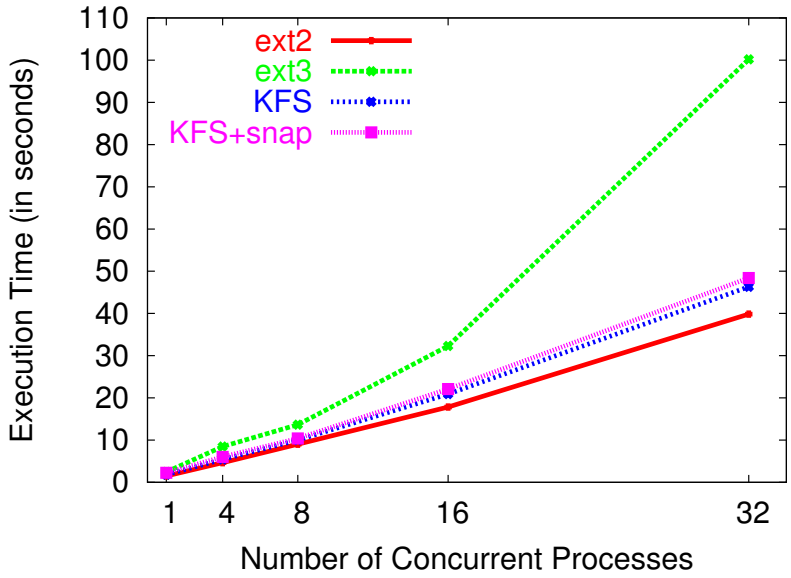

Figure 3: Dbench results. The plotted values represent the execution time necessary to complete a dbench run with the indicated number of clients, for each of the four file system implementations. The values represent the average of 5 consecutive runs.

many Linux distributions, it is the only available file system which has a journalled version, namely, *ext3fs* [Tweedie, 1998]. The ext2 and ext3 file systems are compatible file systems, and share the same on-disk structures. This compatibility suggests that differences in performance are likely to be related to the journaling capability. Details of the experiment are available in [Soares et al., 2003].

The design of KFS allowed us to quickly add such new meta-data consistency policy. Furthermore, KFS's flexibility allows us to pursue how to use this new policy in a finer granularity, i.e., besides being able to achieve meta-data consistency in the presence of crashes, we want to extend the technique to provide file system snapshotting (versioning) and transaction support for a collection of selected files or directories.

4. Exploring Flexibility: Collocation of data and meta-data

Our focus on KFS has been to allow for flexibility at a very fine-grained level, i.e., we want to allow each file in the file system to have a specific, and possibly different, implementation. But as in traditional file systems, our initial prototype implementation used global data structures to represent the file system's meta-data: there was a single object responsible for mapping inodes onto disk location (*RecordMap*, as discussed in Section 2). As our experimentation progressed, we observed some undesirable aspects from using such architecture:

- Lack of scalability as the number of operations that manipulate meta-data increases. When using global structures for maintaining meta-data, all meta-data manipulations in the file system have to use the same structures. These global structures can quickly become points of contention. We have observed for some file system benchmarks that these globals structures were being particularly contented on allocation (e.g., for file creation) and deallocation (e.g., for file removal).
- Flexibility. With global structures for meta-data management, one can easily observe that it is extremely hard for different files or part of the file system tree to use different policies or implementations for meta-data management.
- Disk locality. Another performance issue with using global structures for meta-data is that, in most cases, the file's data and meta-data are located in distinct blocks, possibly

far apart from each other. This implies high seek times for operations which need to access both data and meta-data for files. Additionally, traditional file systems do not use the file system's namespace as hints for applications access patterns. For example, getting directory entries from a particular directory should serve as a hint to the file system that the user will try to access the meta-data of some (or all) of the entries retrieved. Also, it should serve as indication to the file system that meta-data for files in the same directory are probably better off being physically placed together or nearby. The measurements for four workloads presented in [Roselli et al., 2000] showed that the percentage of *stat* calls that follow another *stat* system call to a file from the same directory is around 97%, therefore the directory structure for embedding file attributes in directory entries proposed in [Ganger and Kaashoek, 1997] may provide better performance than storing each file's attribute information with its data blocks.

To solve these problems, we are now studying and implementing a different architecture for meta-data in KFS. The main idea is to remove most of the meta-data global structures in the file system, by having directories responsible for managing meta-data for files in their underlying trees in the file system namespace.

With a *per-directory* management of meta-data, we can more easily address the issues outlined above. Scalability should improve substantially as meta-data structures can be per-directory, which implies that only files on the same partition of the namespace tree are contenting for the same structures. If the namespace tree does not have any anomalies, it will be balanced enough to nicely distribute meta-data manipulation requests.

As for flexibility, the distribution of meta-data management allows for partitions of the namespace tree to have different policies, or even completely different implementations.

An example where this would be useful is the case of directories with few entries. Such a directory might want to optimize disk locality by including the directory data (entries and inode numbers) and the directory entries' meta-data on the same block. Hence, a simple block read would retrieve all necessary information for viewing the contents of such a directory. With such aggressive per-directory meta-data policies and structures, we also address the issue of disk locality.

Even in the case where a single block is not large enough to fit both the directory data and the directory's entries data, it is desirable to store them in neighboring blocks, compacting the data together as much as possible. In this situation, the advantage is not necessarily a reduction of block read operations, but the number of (costly) seeks performed by the disk while preparing to read the necessary blocks. With per-directory management of meta-data, these optimizations become readily feasible and straightforward.

5. Conclusion

This paper presented KFS, a file system designed to be highly adaptable. KFS follows an object-oriented design, with each element in the file system being represented by a different set of objects. There are alternative implementations of file and directories services, each addressing specific file characteristics or access patterns. Developers can add new implementation to address their specific needs, without affecting the performance and functional behavior of other scenarios.

KFS runs on Linux with performance similar to Linux ext2fs's on many workloads. For some benchmarks, KFS performs 15% worse than ext2; we expect that cleaning up and tuning our current prototype (still in its infancy) will be enough to get rid of most of this overhead.

KFS is part of the K42 operating system. The K42 environment allows us to take KFS

flexibility further by exploring K42's support for dynamic adaptation of services. It is possible to switch implementations for specific elements of the file system as they are being actively used. K42's infrastructure for interposition of monitoring objects will allow us to observe access patterns for a file, and adapt the implementation for this file accordingly. In the K42 environment we are also able to achieve better scalability of services in KFS, because we can make use of the highly scalable data structures available in K42.

The flexibility in KFS does not come for free. By breaking up services in different objects to reflect separation of concerns and locality, we may end up with a lot of method invocations to carry out simple work that in traditional file systems reside in a single function call. Allocation and deallocation of the fine-grained objects in KFS come with a cost. But our experience so far indicates that the overhead imposed by the flexibility is easily overthrown by the performance gains achieved by specialized/optimized implementations.

We are initiating work in KFS to explore its flexible design beyond performance gains. Our current goal is to evaluate how much KFS's flexibility simplifies the support of object-based storage.

References

(2004). The K42 Project. http://www.research.ibm.com/K42/.

Anderson, D. (1999). Object based storage devices: A command set proposal. Technical report, National Storage Industry Consortium.

Appavoo, J., Auslander, M., Edelsohn, D., da Silva, D., Krieger, O., Ostrowski, M., Rosenburg, B., Wisniewski, R. W., and Xenidis, J. (2003a). Providing a Linux API on the scalable K42 kernel. In *Freenix*, pages 323–336, San Antonio, TX.

Appavoo, J., Hui, K., Soules, C. A. N., Wisniewski, R. W., da Silva, D., Krieger, O., Auslander, M., Edelsohn, D., Gamsa, B., Ganger, G. R., McKenney, P., Ostrowski, M., Rosenburg, B., Stumm, M., and Xenidis, J. (2003b). Enabling autonomic system software with hot-swapping. *IBM Systems Journal*, 42(1):60–76.

Auslander, M., Franke, H., Gamsa, B., Krieger, O., and Stumm, M. (1997). Customization lite. In *Hot Topics in Operating Systems*, pages 43–48. IEEE.

Blair, J. D. (1998). *Samba: Integrating UNIX and Windows*. Specialized Systems Consultants, Inc.

Card, R., Ts'o, T., and Tweedie, S. (1994). Design and Implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*.

Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., Snir, M., Traversat, B., and Wong, P. (2002). Overview of the MPI-IO parallel I/O interface. In Jin, H., Cortes, T., and Buyya, R., editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 477–487. IEEE.

Custer, H. (1994). *Inside the Windows NT File System*. Microsoft Press.

Ganger, G. and Kaashoek, F. (1997). Embedded inodes and explicit gruopings: Exploiting disk bandwith for small files. In *Proceedings of the 1997 Usenix Annual Technical Conference*, pages 1–17.

Ganger, G. R. and Patt, Y. N. (1994). Metadata Update Performance in File Systems. In *Proceedings of the 1st OSDI*, pages 49–60, Monterey, CA, USA.

Hagmann, R. (1987). Reimplementing the Cedar File System using Logging and Group Commit. In *Proceedings of the 11th SOSP*, pages 155–162, Austin, TX.

Heidemann, J. S. and Popek, G. J. (1994). File system development with stackable layers. *ACM Transaction on Computers*, 12(1):58–89.

Khalid, Y. and Nelson, M. (1993). Extensible file systems in Spring. In *Proceedings of SOSP'93*, pages 1–14. ACM.

Krieger, O. and Stumm, M. (1997). HFS: A performance-oriented flexible filesystem based on build-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321.

linux (2004). The Linux kernel. http://www.kernel.org/.

Mazieres, D. (2001). A toolkit for user-level file systems. In *Proceedings of USENIX'2001*. Usenix.

McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. (1984). A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197.

Mesnier, M., Thereska, E., Ellard, D., Ganger, G. R., and Seltzer, M. (2004). File classification in self-* storage systems. Technical Report CMU-PDL-04-101, Carnegie-Mellon University.

Ousterhout, J. K., Costa, H. D., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. (1985). A Trace-Driven Analisys of the UNIX 4.2 BSD File System. In *Proceeding of the 10th SOSP*, pages 15–24, Orcas Island, Washington.

Roselli, D., Lorch, J. R., and Anderson, T. E. (2000). A comparison of file system workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*, San Diego, CA.

Rosenblum, M. and Ousterhout, J. K. (1992). The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52.

Rosenthal, D. (1990). Evolving the vnode interface. In *Proceedins of USENIX'90*, pages 107–118. Usenix.

Schlosser, S. W. and Ganger, G. R. (2004). MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? In *Proceedings of the 3rd Conference on File and Storage Technlogies (FAST)*. Usenix.

Seltzer, M., Ganger, G., McKusick, M. K., Smith, K., Soules, C., and Stein, C. (2000). Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX Annual Technical Conference*, pages 18–23.

Soares, L. B., Krieger, O., and da Silva, D. (2003). Meta-data snapshotting: A simple mechanism for file system consistency. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, New Orleans, LA.

Soules, C. A. N., Appavoo, J., Hui, K., Wisniewski, R. W., da Silva, D., Ganger, G. R., Krieger, O., Stumm, M., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. (2003). System support for online reconfiguration. In *USENIX*, pages 141–154, San Antonio, TX.

Swartz, K. L. (1997). Adding response time measurement of CIFS file server performance to NetBench. In *Proceedings of the USENIX Windows NT Workshop*, pages 87–94.

Tweedie, S. (1998). Journaling the Linux ext2fs Filesystem. In *LinuxExpo '98*.

Zadok, E. and Nieh, J. (2000). FiST: A language for stackable file systems. In *Proceedings of USENIX'2000*. Usenix.