The Design of New Journaling File Systems: The DualFS Case

Juan Piernas, Toni Cortes, *Member*, *IEEE*, and José M. García, *Member*, *IEEE*

Abstract—This paper describes the foundation, design, implementation, and evaluation of DualFS, a new high-performance journaling file system which has the same consistency guarantees as traditional journaling file systems but a greater performance. DualFS places data and metadata in different *devices* (usually, two partitions of the same storage device) and manages them in very different ways. The *metadata device* is organized as a log-structured file system, whereas the *data device* is organized as groups. The new design allows DualFS not only to recover the consistency quickly after a system crash, but also to improve the overall file system performance. We have evaluated DualFS and we have found that it greatly reduces the total I/O time taken by the file system in most workloads as compared to other file systems (Ext2, Ext3, ReiserFS, XFS, and JFS). The work carried out has also allowed us to draw some lessons which ought to be taken into account when implementing new file systems.

Index Terms—File systems management, secondary storage, operating systems.

1 Introduction

Computing systems rely on several hardware and software elements to fulfill their jobs. Although some environments require specialized elements (i.e., high-performance computing environments), many of them use off-the-shelf elements which must have good performance in order to accomplish computing jobs as quickly as possible.

One of those off-the-shelf elements is the file system. In many computing systems, the file system is a key element which provides a friendly means to store data, executable files, partial and final results, etc. For some computing jobs, the file system used must be fast, others require a quickly recoverable file system, and, for other jobs, both features are important.

File system consistency and performance have to do to a large extent with metadata. With respect to consistency, many file systems [1], [2], [3], [4], [5], [6], [7], [8] use a metadata log approach for fast recovery after a system crash. Other approaches [9], [10] implement some kind of persistent storage which avoids the loss of the most recent updates.

Regarding throughput, file systems should produce as many large sequential accesses as possible. A problem in achieving that kind of access is that metadata requests are usually small and spread across the storage device, which can seriously downgrade file system performance [4], [7], [11], [12], [13]. Moreover, we will show that metadata, even being a very small part of the file system, has a great impact on the overall file system performance. It should, therefore, receive special treatment.

The problems pointed out above indicate that a new file system focused on metadata management is necessary. Current file systems treat data and metadata somewhat differently; however, they are completely different.

This paper describes the foundation, design, implementation, and evaluation of DualFS, a new-generation journaling file system which has the same consistency guarantees as traditional journaling file systems but greater performance.

DualFS is focused on metadata. It separates metadata blocks from data blocks completely and gives them specialized treatment. To the best of our knowledge, no other file system does the same. Metadata blocks are organized as a log-structured file system [5], whereas data blocks are organized in groups (much as in other file systems [3], [6], [14], [15], [16]). The metadata log used by DualFS not only allows DualFS to recover its consistency quickly after a system crash (as occurs in traditional journaling file systems), but also greatly improves metadata operations and, hence, the overall file system performance.

Although the separation between data and metadata is not a new idea, the DualFS design and implementation prove, for the first time, that the separation can significantly improve file systems' performance without requiring extra hardware (previous proposals for the separation of data and metadata, such as the multistructured file system [17], and the interposed request routing [18], use several storage devices).

The rest of the paper is organized as follows: Related work is discussed in Section 2. Section 3 describes the rationale for a new file-system. Section 4 shows an overview of DualFS. Sections 5 and 6 describe the data and metadata devices, respectively. Section 7 explains how DualFS has been tuned by analyzing its sensibility to configuration changes. In Section 8, we describe our benchmarking

- *J. Piernas and J.M. García are with the Departamento de Ingenieria y Tecnologia de Computadores, Facultad de Informatica, Campus de Espinardo, S/N, 30080 Murcia, Spain. E-mail: {piernas, jmgarcia}@ditec.um.es.*
- *T. Cortes is with the Computer Architecture Department, UPC-Campus Nord-C6/118, C/ Jordi Girona 1-3, 08034 Barcelona, Spain. E-mail: toni@ac.upc.edu.*

Manuscript received 6 Apr. 2006; revised 7 July 2006; accepted 7 Aug. 2006; published online 20 Dec. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0137-0406.

TABLE 1
Distribution of Data and Metadata Disk Requests for Different Workloads

	I/O Requests (%)				I/O Time (%)	
Workload	*Data*	*(R/W)*	*Meta-data*	*(R/W)*	*Data*	*Meta-data*
Disk Backup	90.72	(99.94/00.06)	9.28	(71.08/28.92)	86.17	13.83
Root+Mail	28.41	(23.07/76.93)	71.59	(6.45/93.55)	20.47	79.53
Web+FTP	52.11	(63.37/36.63)	47.89	(23.45/76.55)	50.64	49.36
NFS	30.26	(63.06/36.94)	69.74	(27.14/72.86)	57.87	42.13

TABLE 2
Sequentiality of Data and Metadata Requests for Different Workloads

	Same-type Requests		Typeless Requests	
Workload	*Data (%)*	*Meta-data (%)*	*Data (%)*	*Meta-data (%)*
Disk Backup	77.25	1.20	79.92	25.14
Root+Mail	6.01	3.13	6.08	3.14
Web+FTP	42.48	6.43	43.10	7.01
NFS	11.25	10.86	11.47	10.89

methodology and, in Section 9, we present our experimental results. Section 10 concludes the paper.

2 Related Work

In this paper, references to other publications are inserted as they come up in a particular context. Given the organization of our paper, we believe that this is a more natural way of referring to other related proposals. Hence, this section merely makes reference to work that is basic to understanding DualFS.

Journaling file systems [3], [6], [8], [16], [19], [20] are the most common kind of file system found in many current computing systems. DualFS can be considered as a journaling file system but with two important differences at least: DualFS separates data and metadata blocks completely and it writes every metadata block only once (traditional journaling file systems have to write each metadata block twice: once to the log and another time to the final location of the block in the storage device).

Separation of data and metadata has been proposed in the past in order to manage several storage devices and improve performance [17]. Nowadays, this separation is widely used in many distributed/clustered file systems and it is a cornerstone for achieving scalability in those storage systems [18], [21], [22]. DualFS, however, does not need several storage devices.

Finally, a key element in the DualFS design is the structure of the metadata device. The metadata device is organized as a *log-structured file system* [5], [23] which, unlike the original design, is intended to store only metadata blocks.

3 Rationale for a New File System

Since the new journaling file system is based on the separation between data and metadata, we need to know if that separation makes sense. We have analyzed the disk I/O traffic for different common workloads over several days and we have separated data requests from metadata requests. NFS disk traces have been obtained from a server machine running Linux 2.2.14, with 64MB of RAM and an 8GB hard disk, whereas the other traces have been obtained from another server running Linux 2.2.14 with 128MB of RAM and a 3GB hard disk. The results can be seen in Tables 1 and 2.

The first table shows the percentages of data and metadata requests for different workloads and the corresponding percentages of I/O time. The distribution of read and write requests also appears in parentheses.

The second table shows the percentages of sequential requests for the same workloads. For the two columns under the *Same-type requests* title, two requests are considered as sequential when both involve blocks which are both contiguous in disk and of the same type (either data blocks or metadata blocks). For the other two columns, we consider two requests as sequential when both involve blocks which are contiguous in disk, regardless of their types.

The results of the disk backup are expected: Read data requests are predominant and they are mainly sequential. This workload, however, occurs from time to time during a short time period.

The results of the other workloads are more interesting. As we can see, metadata requests represent more than 45 percent of the disk requests and more than 40 percent of the I/O time. Besides, these requests are primarily random writes. These results prove that metadata, even though a very small part of the file system, can have a great impact on the overall file system performance.

Although the results of Tables 1 and 2 are Ext2-specific (they were obtained by instrumenting the Linux kernel and using its default file system), we think they are extensible to other file systems such as Ext3 (which has almost the same structure as Ext2), XFS, and JFS (which share some design principles with Ext2).

Moreover, results achieved are similar to those obtained by Muller and Pasquale [17], Ruemmler and Wilkes [24], and Vogels [25]. There are two main differences between their work and ours. The first one is that they do not calculate the percentage of I/O time taken separately by

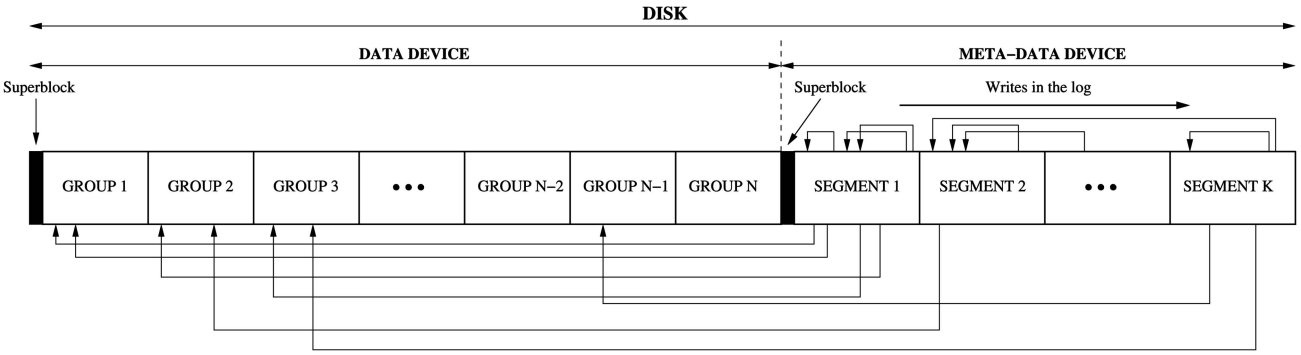

Fig. 1. DualFS overview. Arrows are pointers stored in segments of the metadata device. These pointers are numbers of blocks in both the data device (data blocks) and the metadata device (metadata blocks).

data requests and metadata requests. The second one is that they analyze the sequentiality of file access, but not that of disk requests. I/O time and sequentiality are key information to better understand the performance of a file system.

Our study shows that, if we want to design a new file system to be better than others, we must take into account the special behavior of metadata. Clearly, the new file system must improve metadata writes without damaging either data writes or data and metadata reads.

4 DualFS Overview

The key idea of this new file system is to manage data and metadata in completely different ways, giving a special treatment to metadata. Metadata blocks are located on the *metadata device*, whereas data blocks are located on the *data device*.

It is important to clarify that, by *metadata*, we understand *i-nodes*, *indirect blocks*, *directory "data" blocks*, and *symbolic-links "data" blocks*. Obviously, *bitmaps*, *superblocks*, etc., are also metadata elements. By *data*, we understand the data blocks of regular files.

Files (regular or not) are implemented in DualFS as in other file systems. Every file has an i-node which contains its attributes, and 15 disk addresses: 12 addresses of *direct* blocks and three addresses of *indirect* blocks (the simply, doubly, and triply indirect blocks). In DualFS, the number of i-nodes can grow, as in other file systems such as XFS and JFS.

Direct blocks contain data (or metadata) and their disk addresses refer to the data device, if the file is regular, or to the metadata device, if the file is a directory or a symbolic link. Indirect blocks contain disk addresses of direct blocks or other indirect blocks.

The file system is described by a *superblock* which contains file system parameters (block size, data and metadata devices' sizes, etc.). The superblock is replicated in the data device and throughout the metadata device in order to allow recovery from media failures which corrupt the primary copy of the superblock.

In our current implementation, data and metadata devices are adjacent partitions on the same disk (see Fig. 1). However, other options are also possible. For example, we can use only one disk partition with two areas, one for data blocks and another for metadata blocks, managed by the file system itself. Another possibility is to put the data and metadata devices on different disks, which can improve the performance. This option is similar to putting the log of a journaling file system on a separate disk. However, there is an important difference. Whereas other journaling file systems can exploit the parallelism offered by two disks only for metadata writes, DualFS can exploit the parallelism for both reads and writes of metadata blocks. This is an important advantage for DualFS.

5 The Data Device

The structure of the data device is quite simple. This device only contains data blocks of regular files and uses the concept of *group of data blocks* (similar to the cylinder group concept) for organizing data blocks (see Fig. 1).

The data-block allocation algorithm of DualFS is based on those of Ext2 and Ext3. Since the allocation algorithms used by the three file systems are equivalent, we expect a behavior of DualFS similar to that of Ext2 and Ext3 when the data partition is almost full.

DualFS groups data blocks for several regular files in a per directory basis. When a new directory is created, it is assigned a data-block group in the data device. DualFS specifically selects the group which has the greatest number of free blocks (i.e., the emptiest one). If there are several of them which meet that condition, the group with the smallest number of associated files is selected. Data blocks of regular files created in the same directory are put together in the group assigned to the directory (or in nearby groups if the corresponding group is full). Besides, DualFS tries to allocate all the data blocks of a regular file in such a way as to optimize sequential access.

From the file-system point of view, data blocks are not important for consistency, so they do not receive any special treatment, as metadata does [3], [6], [8]. However, they must be taken into account for security reasons. When a new data block is added to a regular file, DualFS writes it to disk before writing its related metadata blocks. Violating this requirement would not actually damage the consistency, but it could potentially lead to a file containing a previous file's contents after crash recovery, which is a security risk. Ext3 [16] also imposes this order between data and metadata writes by default.

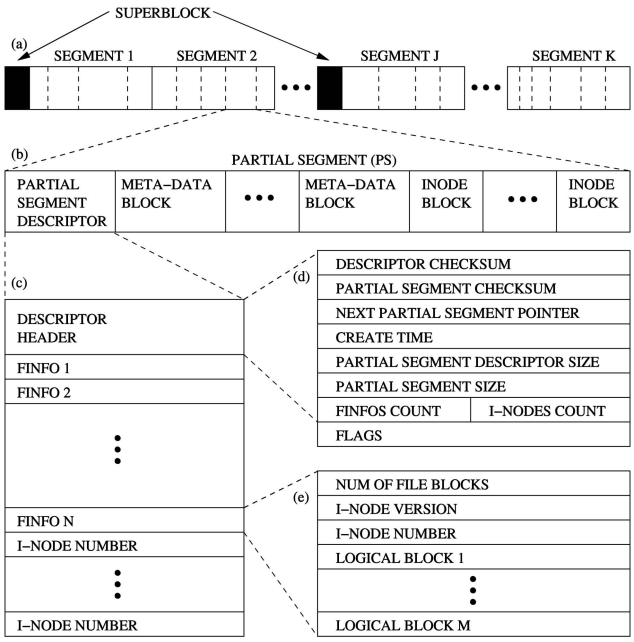

Fig. 2. Structure of the metadata device.

5.1 Directory Affinity

As we have said before, when a new directory is created in DualFS, it is assigned the emptiest group in the data device. Note, however, that when creating a directory tree, DualFS has to select a new group every time a directory is created. This can cause a change of group and, hence, a large seek. However, the new group for the new directory may be only slightly better than that of the parent directory. A better option is to remain in the parent directory's group and to change only when the new group is good enough according to a specific threshold. The latter is called *directory affinity* in DualFS and can greatly improve DualFS performance, as we will see in Section 7.

The directory affinity of DualFS is based on the "directory affinity" concept used by Anderson et al. [18], but with some differences. For example, their directory affinity is a probability (the probability of a new directory being placed on the same server as its parent), whereas it is a threshold in our case. Moreover, their concept is applied to the design of a new network storage while ours is used for the design of a new journaling file systems.

6 The Metadata Device

The structure of the metadata device is fundamental to DualFS since this structure describes almost all of the file system. Moreover, DualFS performance and fast consistency recovery guarantees depend on it.

6.1 Disk Layout

The metadata device is organized as a ***log-structured file system*** [5], that is, there is only one log where all of the metadata blocks are sequentially written (see Fig. 2). Our implementation is based on the BSD-LFS implementation [23]. However, it is important to note that, unlike BSD-LFS, our log does not contain data blocks, only metadata ones. The implementation is also different in some other aspects.

The metadata device is divided into pieces of equal size called ***segments*** (see Fig. 2a). Each segment is 2 MB in size by default, although the segment size can be specified when formatting the file system.

Metadata blocks are written in variable-sized chunks called ***partial segments*** (see Fig. 2b). Partial segments can be as large as a segment, although a segment often accommodates several partial segments which are written asynchronously. For the remainder of this paper, *segment* will be used to refer to the physical partitioning of the metadata device and *partial segment* will be used to refer to the unit of writing.

There is always one, and only one, active partial segment where the last-modified metadata blocks are written to. The active partial segment can be in main memory for some time. This allows multiple metadata updates to be batched into a single log write, which increases the efficiency of the log with respect to the underlying device [5].

A partial segment is a *transaction unit* and must be entirely written to disk to be valid. If a system crash occurs when a partial segment is being written, that partial segment will be dismissed during the file system recovery. Actually, a partial segment is a *compound transaction* because it is the result of several file system operations.

The structure of a partial segment is similar to that of BSD-LFS. The first element of the structure is the ***partial segment descriptor***, which is made up of a descriptor header, *finfo* structures, and i-node numbers (see Fig. 2c). All of the information in the partial segment descriptor is used both for rapidly recovering a crashed file system and for cleaning segments.

The *descriptor header* has information about the partial segment and its structure is shown in Fig. 2d. Since a partial segment is a transaction unit, it is important to know when a partial segment on disk is valid. In order to know that, the descriptor header has two checksum fields, one for the entire descriptor (*descriptor checksum*) and another for the entire partial segment (*partial segment checksum*)—actually, and like BSD-LFS [23], the checksum of the entire partial segment is computed from the first 4 bytes of every partial segment block since checksum computation is very CPU-consuming. These checksums allow DualFS to validate partial segments independently of whether the disk controller reorders writes or not and independently of whether all blocks which make up a partial segment are written to disk or not.

Finfo structures are next. These structures determine the corresponding i-node and logical block number of every block in the partial segment. There is one finfo structure for every file (regular or not) which has at least one metadata block in the partial segment. Fields which make up a finfo are shown in Fig. 2e. Finfo structures ultimately determine the i-node and logical block number of every metadata block in the partial segment. This information will be used by the garbage collection process (the *cleaner*) to reclaim space from the file system.

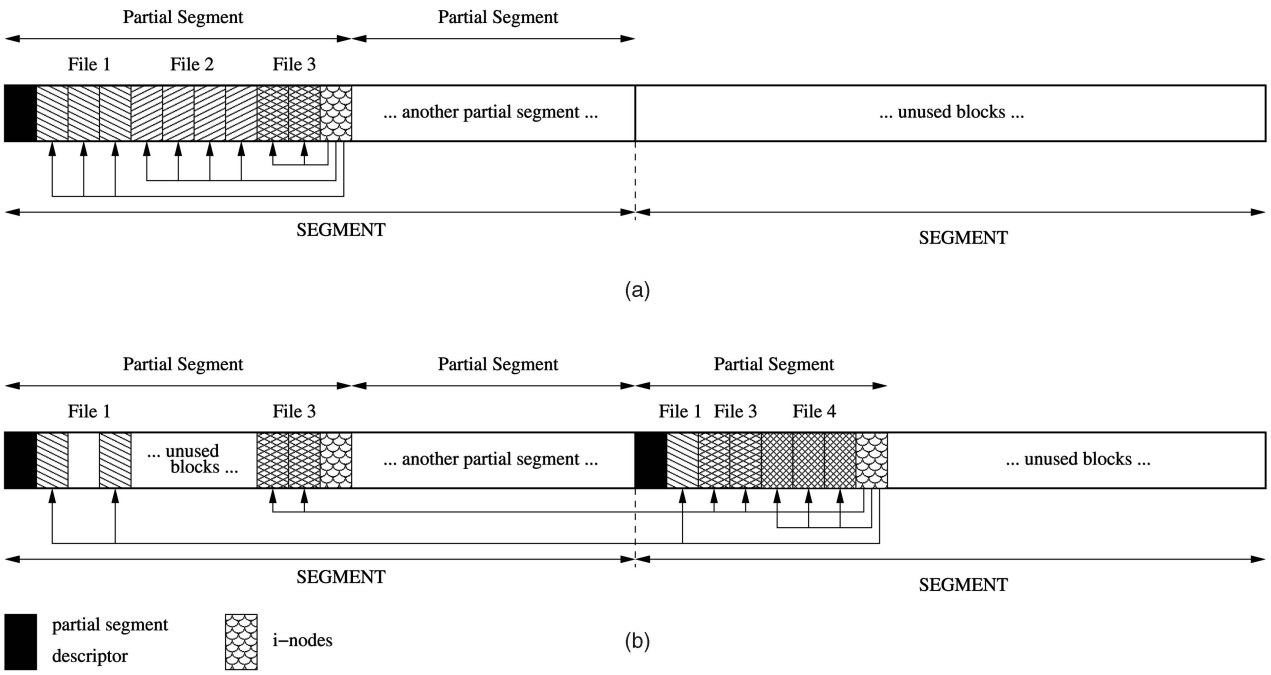

Fig. 3. File allocation in the metadata device. In (a), three files have been written: file1, file2, and file3. In (b), the middle block of file1 has been modified. A new version of it is added to the log, as well as a new version of its corresponding i-node. File2 has been deleted. Although (b) does not show it, a new version of file2 i-node is written in the partial segment of the second segment. Next, two more blocks are appended to file3, causing the blocks and a new version of file3 i-node to be appended to the log. Finally, file4 is created, causing its metadata blocks and i-node to be appended to the log.

Finally, there is a *list of i-node numbers* which contains the numbers of the i-nodes in the partial segment. There are at least as many i-node numbers as finfo structures.

After the partial segment descriptor, there are metadata blocks which belong to several files: indirect blocks, "data" blocks of directories, and "data" blocks of symbolic links.

Finally, at the end of the partial segment, there are several blocks which contain i-nodes. Every block contains as many nonconsecutive i-nodes as it is able to, i.e., the number of blocks is the minimum to contain all the i-nodes in the partial segment.

By using the partial segment structure described above, Fig. 3 shows how metadata blocks of different files are allocated in the metadata device.

6.2 Ifile

Since i-nodes do not have a fixed location on disk, we need an *i-node map* (IMap), i.e., a structure that, given an i-node number, returns the location of the i-node on disk. Our IMap is an array addressed by i-node number. Each IMap entry stores the disk block address of the corresponding i-node and the offset of the i-node inside the block. In DualFS, the i-node map is part of the content of a file called ***IFile***. It is quite obvious that we cannot store the disk address of the IFile's i-node in the IFile itself, so we use the superblock instead.

The structure of our IFile (see Fig. 4) is similar to that of the BSD-LFS IFile [23], but it has two additional elements: the DGDT and the DGBT.

The *data-block group descriptor table* (DGDT) has a role similar to the block group descriptor table of Ext2 [14]. Each data-block group descriptor contains information about the number of i-nodes associated with a group and the number of free data blocks in the group. Both values are useful when DualFS has to assign a new directory to a group. The i-nodes associated with a group are the i-nodes of the directories assigned to the group and the i-nodes of the files created in those directories (except subdirectories).

After the DGDT comes the *data-block group bitmap table* (DGBT). We have a bitmap block for every data-block

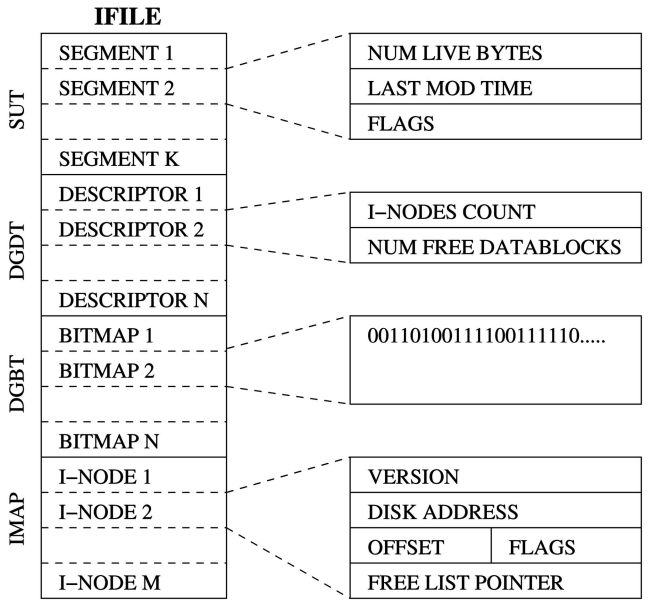

Fig. 4. Structure of the IFile.

group. Therefore, the number of entries in the DGDT table is equal to the number of entries in the DGBT table. Every bitmap indicates which blocks are free and which are busy in a group.

6.3 Cleaner

As a log-structured file system, our metadata device needs a segment cleaner. A segment may contain information still in use (the "live bytes"), but also obsolete information (information superseded by other information in a different segment or information which is no longer valid). A segment that contains live bytes is a dirty segment. Otherwise, it is a clean segment. When the number of clean segments is small, the cleaner can collect the live bytes of several dirty segments and write them in a new segment. Hence, dirty segments will become clean. Obviously, if all dirty segments are full of live bytes, cleaning will be senseless.

Our cleaner is started in two cases: 1) every 5 seconds if the number of clean segments drops below a specific threshold and 2) when we need a new clean segment and all segments are dirty.

At the moment, our attention is not on the cleaner, so we have implemented a simple one, based on Rosenblum and Ousterhout's cleaner [5]. The threshold we have chosen is also very simple (it is a fixed number), though it should depend on both the metadata device size and the workload in a production file system.

In order to select a segment to be cleaned, our cleaner computes $value_i$ for each segment as a cost-benefit function:

$$value_i = \frac{benefit_i}{cost_i} = \frac{(1 - utilization_i) * lastmodtime_i}{1 + utilization_i},$$

where

$$utilization_i = \frac{livebytes_i}{segment\ size}.$$

$livebytes_i$ and $lastmodtime_i$ values are obtained from the *segment usage table* (a data structure in the IFile). This cost-benefit function is the same as that used by Rosenblum and Ousterhout.

The segment with the greatest $value_i$ is cleaned. If the number of clean segments is not large enough after cleaning a segment, the DualFS cleaner recomputes $value_i$ for each segment, cleans the segment with the greatest $value_i$, and so on.

6.4 Log Recovery

Our file system is considered consistent when information about metadata is correct. Like other approaches [6], [8], [26], some loss of data is allowed in the event of a system crash.

Since our metadata device is organized as a log-structured file system, DualFS can quickly recover its consistency by rolling forward from the last checkpoint. Checkpointing is performed every 60 seconds by writing partial segments still in memory, the dirty blocks of the IFile, and the superblock (which contains a reference to the last checkpoint).

Recovering a DualFS file system basically means recovering its IFile. This file is only written to disk at each checkpoint, so its content may be lost at a system crash. During recovery, i-nodes of every partial segment written after the last checkpoint are analyzed to update the IFile. There are three cases for each i-node analyzed:

1. The i-node is new (its IMap entry appears as free). If the new file is regular, we update the DGBT and DGDT tables according to the disk addresses of the file blocks and the SUT table according to the disk addresses of the file's indirect blocks. If the file is not regular, we only have to update the SUT table. Whether the file is regular or not, we must update the corresponding IMap entry of the IFile and several block counters in the superblock.
2. The i-node already exists (its IMap entry appears as busy). We have to find the disk addresses which are different in the previous copy of the i-node and the new one and update the DGBT, DGDT, and SUT tables of the IFile correspondingly. We also have to update the IMap entry in order to reflect the new position of the i-node.
3. The i-node appears as deleted (its reference count field is 0 and its IMap entry appears as busy). This case is similar to the first one, except that data blocks, metadata blocks, and the i-node must be marked as free.

Partial segments are timestamped and checksummed so that the recovering process can easily detect whether a partial segment is valid or not and, hence, when the end of the log is reached.

Note that recovery does not involve redoing or undoing any metadata operation, only updating information in the IFile. It is also clear that the recovery time is proportional to the intercheckpoint interval, not to the file-system size. The same occurs in other journaling file systems.

6.5 Metadata Prefetching

Reading a regular file in DualFS is inefficient because data blocks and their related metadata blocks are a long way from each other and many long seeks are needed.[1] A solution to this problem is metadata prefetching.

The metadata prefetching implemented in DualFS is straightforward: When a metadata block is needed, DualFS reads a group of consecutive metadata blocks from disk where the metadata block is needed. Prefetching is not performed when the metadata block requested is already in memory. The idea is not to force an unnecessary disk I/O request, but to take advantage of a compulsory disk-head movement to the metadata zone. Since all metadata blocks prefetched are consecutive, we also take advantage of the built-in cache of the disk drive.

But, in order to be efficient, our prefetching mechanism needs some kind of metadata locality. As we have seen, DualFS metadata blocks are sequentially written to the log in partial segments. All metadata blocks in a partial segment have been created or modified at the same time. Hence, some kind of relationship between them is expected. Moreover, many relationships between metadata blocks are

1. Since the data and the metadata devices are independent, we could locate them in separate disks, which would solve the read inefficiency.

due to metadata blocks belonging to the same file (e.g., indirect blocks). This kind of temporal and spatial metadata locality present in the DualFS log is what makes our prefetching highly efficient.

Once we have decided when to prefetch, the next step is to decide which and how many metadata blocks must be prefetched. Due to the metadata block order in the DualFS log, the greater part of the metadata blocks which must be prefetched are generally before the needed metadata block. Therefore, our prefetching mechanism has two parameters:

- *size*, which is the amount of metadata blocks prefetched, and
- *after-before ratio*, the percentage of metadata blocks prefetched which are after the requested metadata block.

Later, we will calculate suitable values for these parameters.

It is important to note that, unlike other prefetching methods [27], [28], DualFS prefetching is I/O-time efficient, that is, it does not cause extra I/O requests which can in turn cause long seeks. Also note that our simple prefetching mechanism works because it takes advantage of the unique features of our metadata device.

6.6 Online Metadata Relocation

The metadata prefetching efficiency may deteriorate due to several reasons:

- files are read in an order which is very different from the order in which they were written,
- the read pattern can change over the course of the time, or
- file-system aging.

An inefficient prefetching increases the number of metadata read requests and, hence, the number of disk-head movements between the data zone and the metadata zone. Moreover, it can become counterproductive because it can fill up the buffer cache with useless metadata blocks. In order to avoid prefetching degradation (and to improve its performance in some cases), we have implemented an online metadata relocation mechanism in DualFS which increases temporal and spatial metadata locality.

The metadata relocation works as follows: When a metadata element (i-node, indirect block, directory block, etc.) is read, it is written to the log like any other just-modified metadata element. Note that it does not matter whether the metadata element was already in memory when it was read or if it was read from disk.

Metadata relocation is mainly performed in two cases:

- when reading a regular file (its indirect blocks are relocated) or
- when reading a directory (its "data" and indirect blocks are relocated).

This constant relocation adds more metadata writes. However, these writes are very efficient in DualFS because they are performed sequentially and in big chunks. Therefore, it is expected that this relocation, even when very aggressive, will not increase the total I/O time significantly. We will analyze relocation overhead in the next section.

Since a file is usually read in its entirety [29], the metadata relocation puts together all metadata blocks of the file. The next time the file is read, all its metadata blocks will be together and the prefetching will be very efficient.

The explicit metadata relocation also puts together the metadata blocks of different files read at the same time (i.e., the metadata blocks of a directory and its regular files). If those files are also read later, at the same time and even in the same order, the prefetching will work very well. This assumption in the read order is made by many prefetching techniques [27], [28].

It is important to note that our metadata relocation exposes the relationships between files by writing together the metadata blocks of the files which are read at the same time. Unlike other prefetching techniques [27], [28], this relationship is permanently recorded on disk and it can be exploited by our prefetching mechanism after a system restart.

Besides the explicit relocation, there also exists an implicit metadata relocation which we have not mentioned yet. When a file is read, the *access time* field in its i-node must be updated. If several files are read at the same time, their i-nodes will also be updated at the same time and written together in the log. In this way, when an i-node is read later, the other i-nodes in the same block will also be read. This implicit prefetching is very important since it exploits the temporal locality in the log and can potentially reduce file-open latencies and the overall I/O time.

This implicit metadata relocation can have an effect similar to that achieved by the embedded i-nodes proposed by Ganger and Kaashoek [11]. In their proposal, i-nodes of files in the same directory are put together and stored inside the directory itself. Note, however, that they exploit the spatial locality, whereas we exploit both spatial and temporal localities.

7 DualFS Tuning

In the previous section, we described the general structure of DualFS and the operation of several additional mechanisms implemented for improving file system performance. Some mechanisms, such as prefetching and directory affinity, are configurable by using several values. Others, such as online metadata relocation, can only be either active or inactive and do not have configuration values.

The next subsections summarize the main results of a tuning process carried out on DualFS. Our main purpose has been not only to determine the configuration of the DualFS mechanisms which provides the best performance in general, but also to analyze DualFS's sensibility to configuration changes. The study was performed by using the microbenchmarks described in the next section, although it was necessary to use specific benchmarks in some cases. The system used under test was also the same as that described in the next section.

There exist some aspects of DualFS which have not been analyzed because, from our point of view, they are of little interest (they hardly affect DualFS performance or there is only one valid value for configuration). These aspects, and their configuration values, are: the logical block size (4 KB), the segment size (512 logical blocks or 2 MB), the checkpoint interval (60 seconds), and the data and metadata devices'

sizes (respectively, 90 percent and 10 percent of the storage device).

7.1 Directory Affinity

This is the simplest mechanism because it only affects the data device. We have obtained different results for one and four processes. For one process, a big improvement of around 16 percent in application time occurs for as small a directory affinity as 10 percent (that is, a newly created directory is assigned the emptiest group, instead of its parent directory's group, if the emptiest group has at least 10 percent more free data blocks). For greater directory affinities, improvement is marginal. For four processes, a big improvement also occurs at 10 percent of directory affinity. However, two facts deserve a comment.

The first one is that performance improvement due to directory affinity is not as great for four processes as for one. This is because requests tend to be spread across the disk when there are several processes and directory affinity provides little help for reducing the number of disk head movements.

The second fact is that increasing directory affinity does not always mean an increase in performance. The reason is that the greater the directory affinity, the greater the probability of two or more processes competing for the free space in the same group. This competition translates into more file fragmentation and, hence, worse performance.

The conclusion is that small values of directory affinity can greatly improve DualFS performance, whereas large values may deteriorate it when the number of processes grows. Therefore, we will use a directory affinity of 10 percent in DualFS by default.

7.2 Metadata Prefetching

As we have seen before, metadata prefetching behavior is determined by two parameters: prefetching size and after-before ratio. When there is not enough information (or it is expensive to obtain it from an I/O time perspective), we use default values for both parameters. This occurs in most cases. Otherwise, we use operation-suitable values. This subsection is focused on determining which default value is more convenient for each parameter. Let us analyze both separately.

We have noted that the DualFS performance improves when the prefetching size grows, independently of the after-before ratio. Improvements of up to 75 percent in the application time have been achieved in some microbenchmarks with a prefetching size of 32 blocks. We have also observed that prefetching results are much better when the number of processes increases; this phenomenon will be analyzed in detail in Section 9.

From the results analysis, we have decided to select 16 blocks as the prefetching size. There are several reasons. First, this size matches the DMA transfer size used by Linux (64 KB or 16 blocks of 4 KB). Second, additional performance improvements achieved by greater prefetching sizes are small. And third, medium-size transfers allow DualFS to make better use of the built-in segmented cache of the current disk drives.

With respect to the after-before ratio, there are important differences between the *right* order (that is, all prefetched blocks are before the requested block) and the *wrong* order (the prefetched blocks are after the requested block). The former corresponds to an after-before ratio value of 0 percent, which will be the default value used by DualFS from now on, whereas the latter corresponds to a 100 percent value.

However, we must remark that the results obtained for after-before ratio values of 0 percent and 30 percent are very similar. This is an important fact because it means that it could be profitable to use after-before ratio values somewhat greater than 0 percent in those systems where read patterns may change over the course of the time or do not exactly match with write patterns.

7.3 Online Metadata Relocation

In Section 6.6, we have explained that the online metadata relocation moves metadata blocks during reads. This block movement makes the study of this mechanism more complex than that of directory affinity and metadata prefetching.

As we have already mentioned, relocation adds overhead, but it can improve metadata locality in general and metadata prefetching in particular. However, the relocation can also have unexpected effects; for example, if a process reads files when other (possibly unrelated) processes are reading or writing other files, then relocation can partially break the metadata locality. This is because metadata blocks which are read by the process are mixed with other metadata blocks which are being read or written at the same time. Due to all of the above, the study has been broken down into three parts: overhead, profits provided, and drawbacks.

With respect to the overhead, relocation increases the application time up to 2 percent in benchmarks which involve both data and metadata blocks. The increase goes as far as 9 percent in benchmarks which only involves metadata blocks. Nevertheless, the DualFS performance in this metadata benchmarks is very good as compared to other file systems, so the 9 percent increase is not a problem.

The study of the profits provided by the relocation requires a test without reads and a lot of create/write/delete operations. The aging tools developed by Smith [30] meet our requirements. These tools allow us to age a test file system by replaying a workload which simulates the activity of a real file system over 299 days. After aging a DualFS file system, we read it in its entirety twice and compare the times taken by the first read (without relocation) and the second read (just after relocation).

This benchmark has allowed us to draw two conclusions. The first is that I/O time reduction provided by the implicit relocation of i-nodes is very important. The second is that the online metadata relocation mechanism can guarantee prefetching effectiveness and protect DualFS from aging. Moreover, this explicit relocation reduces the I/O time of DualFS more than 10 percent, which compensates for the aforementioned 2 percent overhead, and makes DualFS the best file system in this benchmark among the file systems compared in Section 9.

Finally, in order to study the possible drawbacks of the metadata relocation, we have implemented a benchmark which simulates a worst-case scenario: There are as many

reads as writes and they are performed at the same time. This benchmark, however, does not represent a common workload because reads are a more frequent operation than writes [25], [29], [31]. The results show that relocation has a limited adverse effect because the application time increases less that 5 percent for one process and this increase is even less when the number of processes grows.

To summarize, we can say that the online metadata relocation is a beneficial mechanism in general, which improves the metadata prefetching effectiveness in many workloads at the expense of a little overhead. Therefore, this mechanism will be active in DualFS by default.

7.4 Cleaner

One of the main drawbacks of a log-structured file system is the cleaner [32]. Since our metadata device is implemented as an LFS, we must evaluate the impact of the cleaner on the DualFS performance.

In order to do that, we have designed a benchmark which produces a lot of half-full dirty segments. The benchmark copies a directory tree and then removes 87.5 percent (7/8) of its regular files. The copy and erasure steps are repeated 40 times in order to make the benchmark take enough time.

This experiment has been carried out for two configurations of DualFS: one that uses the default cleaner and another one that cleans a segment every five seconds. The latter configuration makes the cleaner very intrusive, but it gives us a conservative estimation of the impact of the cleaner on performance.

The results obtained show that the default cleaner hardly runs. However, the intrusive cleaner increases the I/O time by 15 percent. This increase is due to the current implementation of our cleaner, which does the following: 1) Lock any new file system operation, 2) write to disk all partial segments in memory, and 3) clean segments.

Steps 1) and 2) downgrade the file system performance because they lock regular disk operations for a long time. Furthermore, Step 2) is especially problematic because it writes to disk many data blocks which will be deleted soon after (remember that DualFS writes new data blocks to disk before writing their related metadata blocks). If the write order between metadata and new data blocks is omitted, results are much better: The I/O time also increases 15 percent, but it is quite a bit smaller than that obtained when the write order is enforced.

We are convinced that the above performance problems can be mitigated to a large extent by an efficient cleaner implementation which gets rid of the aforementioned Steps 1) and 2). Moreover, we could take advantage of some proposed approaches intended to reduce the cleaner overhead [33]. However, the enhancement has not been carried out because the default DualFS cleaner has run so seldom in all the benchmarks executed (including those described in the next section) that its impact has been *negligible*.

7.5 Partial Segment Write

The last DualFS aspect which deserves an evaluation is the effect of the period length of partial-segment writes from memory to disk. The value of this period affects two conflicting interests: the file system performance and the consistency recovery after a system crash. With respect to the former, a long period may improve the file system throughput because it tends to create big partial segments which are written to disk by means of a few large and sequential requests. With respect to the latter, it is better to have short periods so that the amount of transactions lost at a system crash is small.

From the results, we can conclude that we can obtain a good performance by using a period as short as 5 seconds. For longer periods, the performance improves marginally and not in all cases. This result is important since we can considerably limit the loss of metadata in the event of a system crash without sacrificing throughput.

7.6 Lessons Learned

From the tuning process carried out for DualFS, we have drawn a couple of lessons which could be applied to the design of other file systems. The first one is that the file system must favor temporal locality in disk as much as possible. Although many file systems organize the storage device in groups in order to improve spatial locality and, at the same time, temporal locality, there is not always a straightforward relationship between both kinds of locality. DualFS, however, has an explicit online metadata relocation mechanism which is especially able to improve temporal locality and, besides, to protect the file system against aging (as we have confirmed, the relocation mechanism adds a small overhead, which is indeed advantageous in the long run).

Temporal locality can provide many benefits. First, it can cause shorter disk seeks, as spatial locality does. Second, it makes it possible to implement efficient prefetching mechanisms, much like in DualFS. And third, it could improve the performance of some disk schedulers [34].

The second lesson learned is that the group structure and the directories' allocation policies used by many file systems can produce many long seeks when traversing a directory tree, especially if there are several processes. The directory affinity implemented by DualFS is a simple allocation policy which tries to place a subdirectory in the same group as its parent, which improves spatial locality. Other more complex policies, such as the allocator designed by Orlov [35], pursue a similar aim by favoring the placement of directories close to their parent.

8 Methodology for Experiments

This section describes the evaluation process of DualFS. We have used both microbenchmarks and macrobenchmarks for different configurations, using the Linux kernel 2.4.19. We have compared DualFS against Ext2 [14], the default file system in Linux, and four journaling file systems: Ext3 [16], XFS [6], [36], JFS [3], and ReiserFS [20]. Ext2 is not a journaling file system, but it has been included because it is a widely used and well-understood file system.

Bryant et al. [37] compared the above five file systems by using several benchmarks on three different systems, ranging in size from a single-user workstation to a 28-processor ccNUMA machine. However, there are some important differences between their work and ours. First, we evaluate a next-generation journaling file system

(DualFS). Second, we report results for some industry-standard benchmarks (SPECweb99 and TPC-C). Third, we use microbenchmarks which are able to clearly expose performance differences between file systems (in their single-user workstation system, for example, only one benchmark was able to show performance differences). And finally, we also report I/O time at disk level (this measurement is important because it reflects the behavior of every file system as seen by the disk drive).

8.1 Microbenchmarks

We have designed seven microbenchmarks intended to discover the strengths and weaknesses of every file system:

- *read-meta (r-m)*: Find files larger than 2 KB in a directory tree.
- *read-data-meta (r-dm)*: Read all of the regular files in a directory tree.
- *write-meta (w-m)*: Untar an archive which contains a directory tree with empty files.
- *write-data-meta (w-dm)*: The same as *w-m*, but with nonempty files.
- *read-write-meta (rw-m)*: Copy a directory tree with empty files.
- *read-write-data-meta (rw-dm)*: The same as *rw-m*, but with nonempty files.
- *delete (del)*: Delete a directory tree.

In all cases, the directory tree has four subdirectories, each one with a copy of a clean Linux 2.4.19 source tree. In the "write-meta" and "read-write-meta" benchmarks, we have truncated to zero all regular files in the directory tree. In the "write-meta" and "write-data-meta" tests, the untarred archive is in a file system which is not being analyzed. All tests have been run five times for one and four processes. When there are four processes, each works on one of the four copies of the Linux source tree.

8.2 Macrobenchmarks

Next, we list the benchmarks we have performed to study the viability of our proposal. Note that we have chosen environments that are currently representative:

- *Kernel Compilation for 1 Process (KC-1P)*: Resolve dependencies (*make dep*) and compile the Linux kernel 2.4.19, given a common configuration. Kernel and modules compilation phases are made for one process (*make bzImage* and *make modules*).
- *Kernel Compilation for 4 Processes (KC-4P)*: The same as before, but just for four processes (*make -j4 bzImage*, and *make -j4 modules*).
- *SPECweb99 (SW99)*: The SPECweb99 benchmark [38]. We have used two machines: a server, with the file system to be analyzed, and a client.
- *PostMark (PM)*: The PostMark benchmark, which models the workload seen by ISPs under heavy load [39]. We have run our experiments using version 1.5 of the benchmark. In our case, the benchmark initially creates 150,000 files with a size range of 512 bytes to 16 KB, spread across 150 subdirectories. Then, it performs 20,000 transactions with no bias toward any particular transaction type and with a transaction block of 512 bytes.
- *TPCC-UVA (TPC-C)*: An implementation of the TPC-C benchmark. Due to system limitations, we have only used three warehouses. The benchmark is run with an initial 30 minutes warm-up stage and a subsequent measure time of 2 hours.

TABLE 3
System under Test

Processor	Two 450 MHz Pentium III
Memory	256 MB, PC100 SDRAM
Disk	**Test disk**: One 4 GB 5,400 RPM Seagate ST-34310A IDE disk. **System disk** (operating system, and traces): One 4GB 10,000 RPM FUJITSU MAC3045SC SCSI disk.
OS	Linux 2.4.19

8.3 Tested Configurations

All benchmarks have been run for the six file systems shown below. Mount options have been selected following recommendations by Bryant et al. [37]:

- Ext2, without any special mount option.
- Ext3, with "`data=ordered`" mount option.
- XFS, with "`logbufs=8,osyncisdsync`" mount options.
- JFS, without any special mount option.
- ReiserFS, with "`notail`" mount option.
- DualFS, with metadata prefetching, online metadata relocation, and directory affinity.

The versions of Ext2, Ext3, and ReiserFS are those found in a standard Linux kernel 2.4.19. The XFS version is 1.1 and the JFS version is 1.1.1.

All file systems are on one IDE disk and use a logical block size of 4 KB. DualFS uses two adjacent partitions, whereas the other file systems only use one disk partition. The DualFS metadata device is always on the outer partition since this partition is faster than the inner one and its size is 10 percent of the total disk space. The inner partition is the data device. DualFS also has a prefetching size of 16 blocks and a directory affinity of 10 percent.

8.4 System under Test

All tests have been done on the same machine, whose configuration is shown in Table 3. In order to trace disk I/O activity, we have instrumented the operating system to record when a request starts and finishes.

9 Experimental Results

In order to better understand the different file systems, we show two performance results for each file system in every benchmark:

- *Disk I/O time*: The total time taken for all disk I/O operations.
- *Performance*: The performance achieved by the file system in the benchmark.

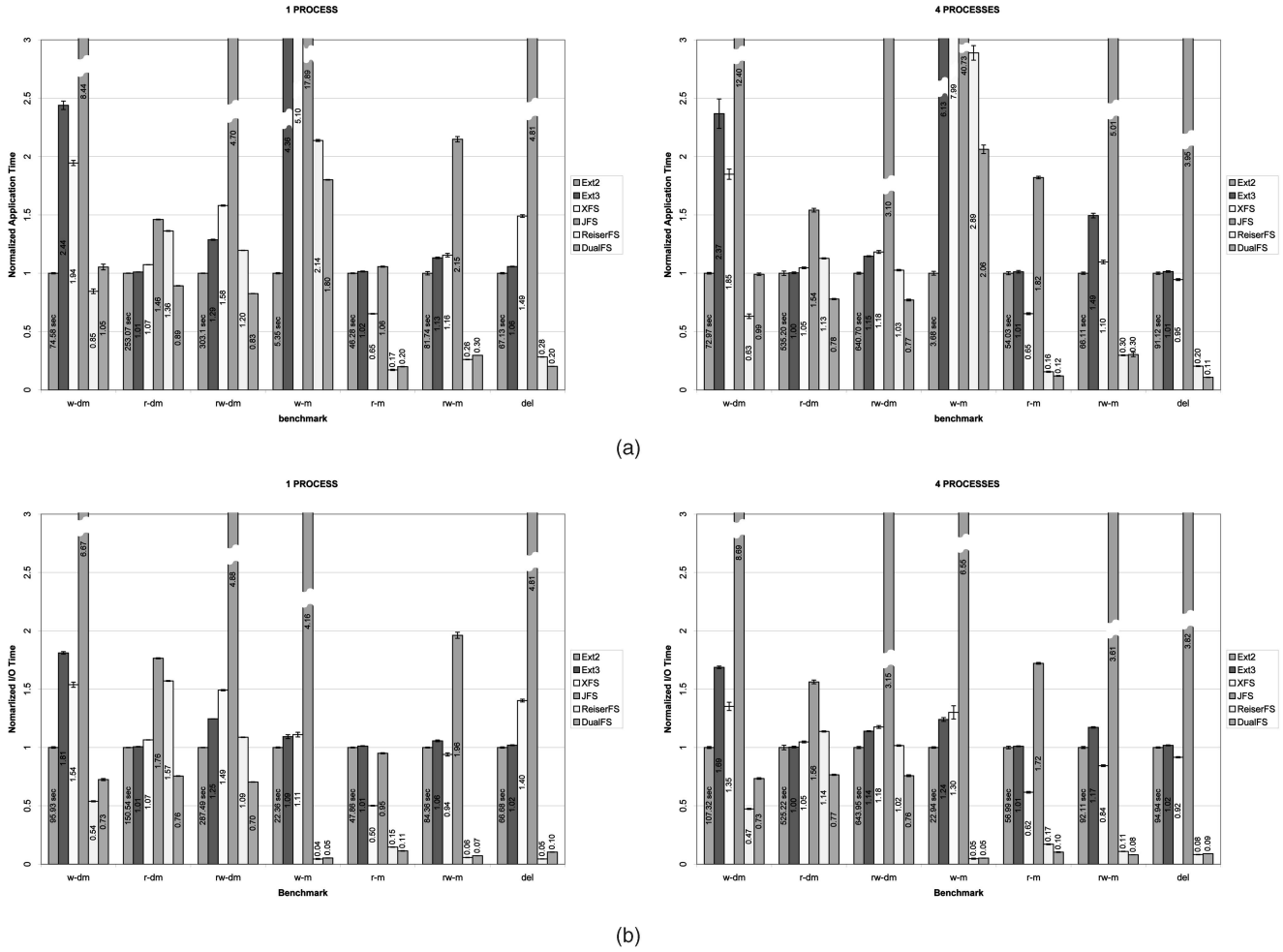

Fig. 5. Microbenchmarks results. (a) Application time. (b) Disk I/O time.

The *performance* result is the *application time* except for the SPECweb99 and TPC-C benchmarks. Since these macrobenchmarks take a fixed time specified by the benchmark itself, we will use the benchmark metrics (number of simultaneous connections for SPECweb99 and tpmC for TPC-C) as throughput measurements.

We could give the application time only, but, given that some macrobenchmarks (e.g., compilation) are CPU-bound in our system, we find I/O time more useful in those cases. The total I/O time can give us an idea of to what extent the storage system can be loaded. A file system that loads a disk less than others makes it possible to increase the number of applications which perform disk operations concurrently.

The figures listed below show the confidence intervals for the mean as error bars for a 95 percent confidence level. The number inside each bar is the height of the bar, which is a value normalized with respect to Ext2. For Ext2, the height is always 1, so we have written the absolute value for Ext2 inside each Ext2 bar, which is useful for comparison purposes between figures.

Finally, it is important to note that all benchmarks have been run with a cold file system cache (the computer is restarted after every test run).

9.1 Microbenchmarks

Fig. 5 shows the microbenchmarks' results. A quick review shows that DualFS has the best disk I/O and application times in general, in both write and read operations, and that JFS has the worst performance. Only ReiserFS is clearly better than DualFS in the *write-data-meta* benchmark and it is even better when there are four processes. However, ReiserFS performance is very poor in the *read-data-meta* case. This is a serious problem for ReiserFS given that reads are a more frequent operation than writes [25], [29]. In order to understand these results, we must explain some ReiserFS features.

ReiserFS does not use the block group or cylinder group concepts like the other file systems analyzed. Instead, ReiserFS uses an allocation algorithm which allocates blocks almost sequentially when the file system is empty. This allocation starts at the beginning of the file system, after the last block of the metadata log. Since many blocks are allocated sequentially, they are also written sequentially. The other file systems, however, have data blocks spread across different groups which take up the entire file system, so writes are not as efficient as in ReiserFS. This explains the good performance of ReiserFS in the *write-data-meta* test.

The above also explains why ReiserFS is not so bad in the *read-data-meta* test when there are four processes. Since the

TABLE 4
Results of the *read-meta* Test

File System	I/O Time		Increase (%)
	1 process	4 processes	
Ext2	47.86 (0.35)	56.99 (1.13)	19.08
Ext3	48.45 (0.29)	57.59 (0.35)	18.86
XFS	24.00 (0.66)	35.13 (1.15)	46.38
JFS	45.49 (0.46)	98.10 (0.44)	115.65
ReiserFS	7.02 (1.44)	9.77 (2.32)	39.17
DualFS	5.46 (2.00)	5.90 (3.33)	8.06

directory tree read by the processes is small and it is created in an empty file system, all its blocks are together at the beginning of the file system. This makes processes' read requests cause small seeks when processes read the directory tree. The same does not occur in the other file systems, where the four processes cause long seeks because they read blocks which are in different groups spread across the entire disk.

Another interesting point in the *write-data-meta* benchmark is the performance of ReiserFS and DualFS with respect to Ext2. Although the disk I/O times of both file systems are much better than that of Ext2, the application times are not so good. This indicates that Ext2 achieves a better overlap between I/O and CPU operations because it neither has a journal nor forces a metadata write order.

The good performance of DualFS is specially remarkable in the *read-data-meta* benchmark, where DualFS is up to 35 percent faster than ReiserFS. When compared with previous versions of DualFS [40], we can see that this performance is mainly provided by the metadata prefetching and directory affinity mechanisms, which, respectively, reduce the number of long seeks between data and metadata partitions and between data-block groups within the data partition.

In all the metadata-only benchmarks but *write-meta*, the distinguished winners (regarding the application time) are ReiserFS and DualFS. And, they are the absolute winners, taking into account the disk I/O time. Also note that DualFS is the best when the number of processes is four.

The problem of the *write-meta* benchmark is that it is CPU-bound because all modified metadata blocks fit in main memory. In this benchmark, Ext2 is very efficient, whereas the journaling file systems are big consumers of CPU time in general. Even then, DualFS is the best of the five. Regarding the disk I/O time, ReiserFS and DualFS are the best, as expected, because they write metadata blocks to disk sequentially.

In the *read-meta* case, ReiserFS and DualFS have the best performance because they read metadata blocks which are very close. Ext2, Ext3, XFS, and JFS, however, have metadata blocks spread across the storage device, which causes long disk-head movements. Note that the DualFS performance is even better when there are four processes.

This increase in DualFS performance when the number of processes goes from one to four is due to the metadata prefetching. Indeed, prefetching makes DualFS scalable with the number of processes. Table 4 shows the I/O time (in seconds) for the six file systems studied and for one and four processes (the value in parentheses is the confidence interval given as percentage of the mean). Whereas Ext2, Ext3, XFS, JFS, and ReiserFS significantly increase the total I/O time when the number of processes goes from one to four, DualFS increases the I/O time slightly.

For one process, the high metadata locality in the DualFS log and the implicit prefetching performed by the disk drive (through the read-ahead mechanism) make the difference between DualFS and the other file systems. ReiserFS also takes advantage of the same disk read-ahead mechanism.

However, the implicit prefetching performed by the disk drive is less effective if the number of processes is two or greater. When there are four processes, the disk heads constantly go from track to track because each process works with a different area of the metadata device. When the disk drive reads a new track, the previous read track is evicted from the built-in disk cache and its metadata blocks are discarded before being requested by the process which caused the read of the track.

The explicit prefetching performed by DualFS solves the above problem by copying metadata blocks from the built-in disk cache to the buffer cache in main memory before being evicted. Metadata blocks can stay in the buffer cache for a long time, whereas metadata blocks in the built-in cache will be evicted soon, when the disk drive reads another track.

Another remarkable point in the *read-meta* benchmark is the XFS performance. Although XFS has metadata blocks spread across the storage device like Ext2 and Ext3, its performance is much better. We have analyzed XFS disk I/O traces and we have found out that XFS does not update the "atime" of directories by default. The absence of metadata writes in XFS reduces the total I/O time because there are fewer disk operations and because the average time of the read requests is smaller. JFS does not update the "atime" of directories either, but that does not appear to reduce its I/O time significantly.

In the last benchmark, *del*, the XFS behavior is odd again. For one process, it has a very bad performance. However, the performance improves when there are four processes. The other file systems have the behavior we expect.

Finally, note the great performance of DualFS in the *read-data-meta* and *read-meta* benchmarks despite the online metadata relocation.

9.2 Macrobenchmarks

The results of macrobenchmarks are shown in Fig. 6. Since benchmark metrics are different, we have shown the relative application performance with respect to Ext2 for every file system.

As we can see, the only I/O-bound benchmark is PostMark (that is, benchmark results agree with I/O time results). The other four benchmarks are CPU-bound in our system and all file systems consequently have a similar performance. Nevertheless, DualFS is usually a little better than the other file systems.

The reason to include the CPU-bound benchmarks is that they are very common for system evaluations. Moreover, the I/O time can be important in a more powerful system running one of those benchmarks. Hence, it is at least

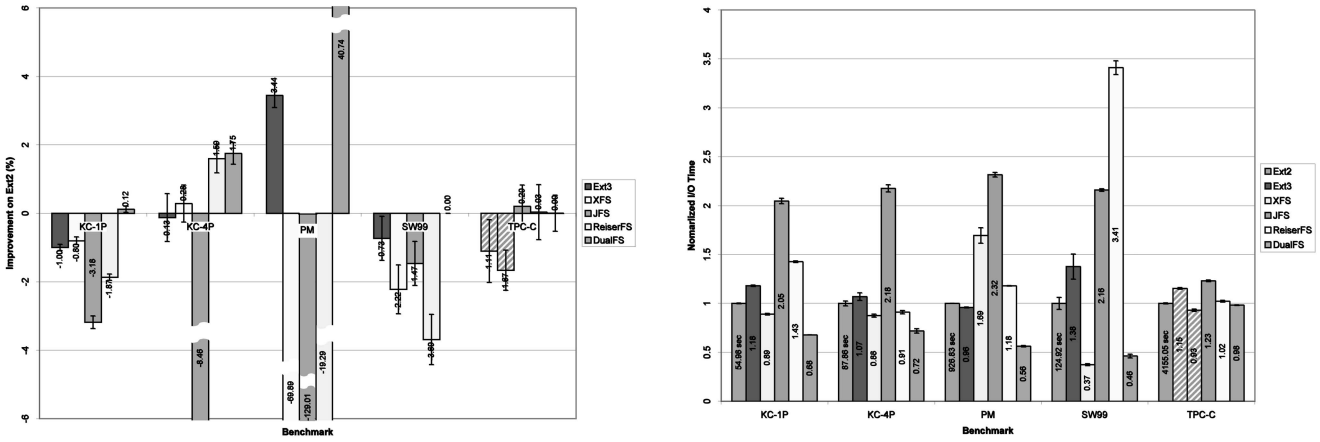

Fig. 6. Macrobenchmarks' results. The left figure shows the application throughput improvement achieved by each file system with respect to Ext2. The right figure shows the disk I/O time normalized with respect to Ext2. In the TPC-C benchmark, Ext3 and XFS bars are striped because they do not meet TPC-C benchmark requirements.

interesting to analyze the I/O time taken by each file system in the four CPU-bound benchmarks.

From the disk I/O time point of view, DualFS has the best throughput. Only XFS is better than DualFS in the SPECweb99 and TPC-C benchmarks. However, we must, respectively, take into account that XFS neither updates the access time of directory i-nodes nor meets the TPC-C benchmark requirements (specifically, it does not meet the response time constraints for new-order and order-status transactions).

In order to explain this superiority of DualFS over the other file systems, we have analyzed the disk I/O traces obtained and we have found that performance differences between file systems are mainly due to writes (see Fig. 7). There are a lot of write operations in these tests and DualFS is the file system which better carries them out. For JFS, however, these performance differences are also due to reads, which take longer than in the other file systems.

Internet System Providers should pay special attention to the results achieved by DualFS in the PostMark benchmark. In this test, DualFS achieves 60 percent more transactions per second than Ext2 and Ext3, twice as many transactions per second as ReiserFS, almost three times as many transactions per second as XFS, and four times as many transactions per second as JFS. Although there are specific file systems for Internet workloads [41], note that these results are achieved by a general-purpose file system, DualFS.

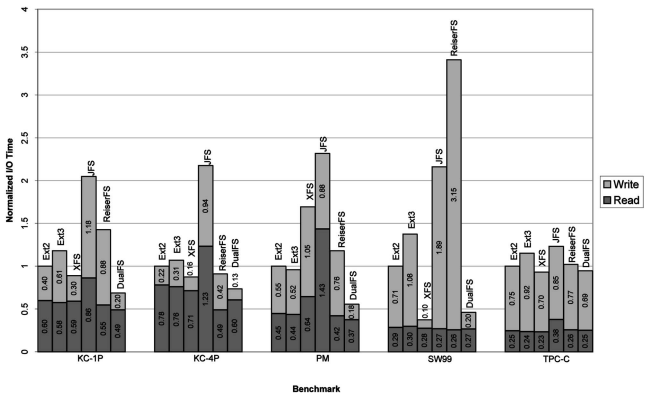

Fig. 7. Distribution of read time and write time in macrobenchmarks.

9.3 Lessons Learned

After comparing DualFS with other file systems, there are some lessons that can be learned. The first one is that the separation between data and metadata, along with suitable storage structures for each one, facilitates the implementation of more efficient file systems, as the above experimental results show.

The second lesson is that the group structure used by Ext2, Ext3, XFS, and JFS can greatly downgrade file system performance when there are several processes performing metadata operations. The reason is that there are a lot of long seeks caused by the reading and writing of small elements, such as metadata. In order to solve this problem, it is preferable to place metadata apart in a small device with a proper structure which optimizes writes, the most frequent operation on metadata. The DualFS's metadata device meets these requirements.

Finally, the last lesson learned is that it is important to move from the built-in disk cache to the main memory everything which might be needed soon after, especially when there are a lot of processes. Since the built-in disk cache is relatively small, blocks read from disk are soon evicted and must be read again if they are needed later. The usefulness of this block movement can be guaranteed with a high temporal locality in disk. The prefetching and online relocation mechanisms implemented by DualFS achieve this goal for metadata. The other file systems do not have anything similar and their performances fall down in metadata benchmarks when the number of processes rises.

10 Conclusions

Improving file system performance is important for a wide variety of systems, from general purpose systems to more specialized high-performance systems. Many high-performance systems, for example, rely on off-the-shelf file

systems to store final and partial results and to resume failed computation.

In this paper, we have described DualFS, which proves, for the first time, that a new journaling file-system design based on data and metadata separation and special metadata management is not only possible but also desirable. DualFS design is focused on metadata management. Our new file system organizes metadata as a log-structured file system, while data blocks are organized in groups, much as in other file systems.

The new journaling file system has turned out to be very efficient. Through an extensive set of micro and macro-benchmarks, we have evaluated six different journaling and nonjournaling file systems (Ext2, Ext3, XFS, JFS, ReiserFS, and DualFS) and the experimental results obtained have shown that DualFS is the best journaling file system in general.

ACKNOWLEDGMENTS

This work has been supported by the Spanish Ministry of Science and Technology and the European Union (FEDER funds) under the TIC2003-08154-C06-03 and TIN2004-07739-C02-01 CICYT grants.

REFERENCES

[1] S. Chutani, T. Anderson, M.L. Kazar, B.W. Leverett, W.A. Mason, and R. Sidebotham, "The Episode File System," *Proc. Winter 1992 USENIX Conf.,* pp. 43-60, Jan. 1992.

[2] R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *Proc. 11th ACM Symp. Operating Systems Principles (SOSP),* pp. 155-162, Nov. 1987.

[3] IBM Corp., "JFS for Linux," 2006, http://oss.software.ibm.com/jfs.

[4] J.K. Peacock, A. Kamaraju, and S. Agrawal, "Fast Consistency Checking for the Solaris File System," *Proc. 1998 USENIX Ann. Technical Conf.,* pp. 77-89, June 1998.

[5] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems,* vol. 10, no. 1, pp. 26-52, Feb. 1992.

[6] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *Proc. 1996 USENIX Ann. Technical Conf.,* pp. 1-14, Jan. 1996.

[7] U. Vahalia, C.G. Gray, and D. Ting, "Metadata Logging in an NFS Server," *Proc. 1995 USENIX Technical Conf.,* pp. 265-276, Jan. 1995.

[8] Veritas Software, "The VERITAS File System (VxFS)," 1995, http://www.veritas.com/products/.

[9] P.M. Chen, W.T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," *Proc.f Architectural Support for Programming Languages and Operating Systems (ASPLOS),* pp. 74-83, Oct. 1996.

[10] Y. Hu, Q. Yang, and T. Nightingale, "Rapid-Cache—A Reliable and Inexpensive Write Cache for Disk I/O Systems," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 204-213, Jan. 1999.

[11] G.R. Ganger and M.F. Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," *Proc. 1997 USENIX Ann. Technical Conf.,* pp. 1-17, Jan. 1997.

[12] G.R. Ganger and Y. Patt, "Metadata Update Performance in File Systems," *Proc. First USENIX Symp. Operating Systems Design and Implementation (OSDI),* pp. 49-60, Nov. 1994.

[13] M.I. Seltzer, G.R. Ganger, M.K. McKusick, K.A. Smith, C.A. Soules, and C.A. Stein, "Journaling versus Soft Updates: Asynchronous Meta-Data Protection in File Systems," *Proc. 2000 USENIX Ann. Technical Conf.,* June 2000.

[14] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," *Proc. First Dutch Int'l Symp. Linux,* Dec. 1994.

[15] M. McKusick, M. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX," *ACM Trans. Computer Systems,* vol. 2, no. 3, pp. 181-197, Aug. 1984.

[16] S. Tweedie, "Journaling the Linux ext2fs Filesystem," *Proc. LinuxExpo '98,* 1998.

[17] K. Muller and J. Pasquale, "A High Performance Multi-Structured File System Design," *Proc. 13th ACM Symp. Operating Systems Principles (SOSP),* pp. 56-67, Oct. 1991.

[18] D.C. Anderson, J.S. Chase, and A.M. Vahdat, "Interposed Request Routing for Scalable Network Storage," *Proc. Fourth USENIX Symp. Operating Systems Design and Implementation (OSDI),* pp. 259-272, Oct. 2000.

[19] M.E. Russinovich and D.A. Solomon, *Microsoft Windows Internals,* fourth ed. Microsoft Press, 2005.

[20] Namesys, "ReiserFS File System," 2006, http://www.namesys.com.

[21] P.H. Carns, W.B. Ligon III, R.B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," *Proc. Fourth Ann. Linux Showcase and Conf.,* pp. 317-327, Oct. 2000.

[22] F. Schmuck and R. Haskin, "Gpfs: A Shared-Disk File System for Large Computing Clusters," *Proc. 2002 USENIX Conf. File and Storage Technologies (FAST),* pp. 231-244, Jan. 2002.

[23] M. Seltzer, K. Bostic, M.K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proc. Winter 1993 USENIX Conf.,* pp. 307-326, Jan. 1993.

[24] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proc. 1993 USENIX Winter Technical Conf.,* pp. 405-420, Jan. 1993.

[25] W. Vogels, "File System Usage in Windows NT 4.0," *ACM SIGOPS Operating Systems Rev.,* vol. 34, no. 5, pp. 93-109, Dec. 1999.

[26] M.K. McKusick and G.R. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proc. 1999 USENIX Ann. Technical Conf.,* pp. 1-17, June 1999.

[27] T.M. Kroeger and D.D.E. Long, "Design and Implementation of a Predictive File Prefetching Algorithm," *Proc. 2001 USENIX Ann. Technical Conf.,* pp. 319-328, June 2001.

[28] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching," *Proc. 1997 USENIX Ann. Technical Conf.,* pp. 275-288, 1997.

[29] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout, "Measurements of a Distributed File System," *Proc. 13th ACM Symp. Operating Systems Principles (SOSP),* pp. 198-212, 1991.

[30] K.A. Smith, "Aging Tools and Trace File," 1996, http://www.eecs.harvard.edu/keith/usenix96.

[31] D. Roselli, J.R. Lorch, and T.E. Anderson, "A Comparison of File System Workloads," *Proc. 2000 USENIX Ann. Technical Conf.,* pp. 41-54, June 2000.

[32] M. Seltzer, K.A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File System Logging versus Clustering: A Performance Comparison," *Proc. 1995 USENIX Technical Conf.,* pp. 249-264, Jan. 1995.

[33] J. Wang and Y. Hu, "WOLF—A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems," *Proc. 2002 USENIX Conf. File and Storage Technologies (FAST),* Jan. 2002.

[34] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," *Proc. 18th ACM Symp. Operating Systems Principles,* pp. 117-130, 2001.

[35] I. Dowse and D. Malone, "Recent Filesystems Optimisations in FreeBSD," *Proc. FREENIX Track: 2002 USENIX Ann. Technical Conf.,* pp. 245-258, June 2002.

[36] Silicon Graphics Inc., "Linux XFS," 2006, http://linux-xfs.sgi.com/projects/xfs.

[37] R. Bryant, R. Forester, and J. Hawkes, "Filesystem Performance and Scalability in Linux 2.4.17," *Proc. FREENIX Track: 2002 USENIX Ann. Technical Conf.,* pp. 259-274, June 2002.

[38] Standard Performance Evaluation Corp., "The SPECweb99 Benchmark," 1999, http://www.spec.org.

[39] J. Katcher, "PostMark: A New File System Benchmark," Technical Report TR3022, Network Appliance Inc., Oct. 1997.

[40] J. Piernas, T. Cortes, and J.M. García, "DualFS: A New Journaling File System without Meta-Data Duplication," *Proc. 16th Ann. ACM Int'l Conf. Supercomputing (ICS),* pp. 137-146, June 2002.

[41] E. Shriver, E. Gabber, L. Huang, and C. Stein, "Storage Management for Web Proxies," *Proc. 2001 USENIX Ann. Technical Conf.,* pp. 203-216, June 2001.

Juan Piernas received the MS degree in computer science in 1994 and the PhD degree, also in computer science, in 2004 (both from the Universidad de Murcia). He has been an assistant professor at the Universidad de Murcia (Spain) since 1995 and vice-dean of Quality and European Convergence since 2004. His main research concentrates on operating systems, storage systems, and distributed systems. He has published some refereed papers in different journals and conferences in these fields. He has also participated in several Spanish funded projects.

Toni Cortes received the MS degree in computer science in 1992 and the PhD degree, also in computer science, in 1997 (both from the Universitat Politècnica de Catalunya). He is an associate professor at the Universitat Politècnica de Catalunya and the manager of the storage-system group at the BSC. His research concentrates on storage systems, programming models for distributed systems, and operating systems. He has published more than 40 papers in international conferences and journals. He has also participated in European funded projects such as Paros, Nanos, and POP as well as cooperation projects with companies such as IBM. He is currently the coordinator of the single-system image area in the IEEE Technical Committee on Scalable Computing. He is a member of the IEEE.

José M. García received the MS and the PhD degrees in electrical engineering from the Technical University of Valencia, Spain, in 1987 and 1991, respectively. Dr. García is a full professor in the Department of Computer Engineering at the Universidad de Murcia, Spain, and also the head of the Research Group on Parallel Computer Architecture. At present, he is serving as the dean of the School of Computer Science at the Universidad de Murcia. His current research interests lie in high-performance coherence protocols for chip multiprocessor (CMP) and shared-memory multiprocessor systems and high-speed interconnection networks. He has published more than 80 refereed papers in different journals and conferences in these fields. Professor García is a member of HiPEAC, the European Network of Excellence on High-Performance Embedded Architecture and Compilation. He is also a member of several international associations, such as the IEEE and the ACM, and also a member of some European associations (Euromicro and ATI).