Controlling your PLACE in the File System with Gray-box Techniques

James A. Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison

Abstract

We present the design and implementation of PLACE, a gray-box library for controlling file layout on top of FFS-like file systems. PLACE exploits its knowledge of FFS layout policies to let users place files and directories into specific and localized portions of disk. Applications can use PLACE to collocate files that exhibit temporal locality of access, thus improving performance. Through a series of microbenchmarks, we analyze the overheads of controlling file layout on top of the file system, showing that the overheads are not prohibitive, and also discuss the limitations of our approach. Finally, we demonstrate the utility of PLACE through two case studies: we demonstrate the potential of file layout rearrangement in a web-server environment, and we build a benchmarking tool that exploits control over file placement to quickly extract low-level details from the disk system. In the traditional gray-box manner, the PLACE library achieves these ends entirely at user level, without changing a single line of operating system source code.

1 Introduction

Creators of high-performance I/O-intensive applications, including database management systems and web servers, have long yearned for control over the placement of their data on disk [26]. Proper data allocation can exploit locality of access within a particular workload, increasing disk efficiency and thereby improving overall performance.

However, many file systems do not provide the explicit controls that are needed by applications to affect their desired file layouts. For example, UNIX file systems based on the Berkeley Fast File System (FFS) [13] group files by a set of heuristics, specifically trying to group inodes and data blocks of files that reside in the same directory. Applications that wish to have full control over layout traditionally have avoided using file systems altogether, thus relinquishing convenience for control.

Gray-box techniques [1, 4] are a promising approach that can be used to gather information about and exert control over systems that do not export the necessary interfaces to do so. By treating a system as a *gray box*, one assumes some general knowledge of how the system behaves or is implemented; such knowledge, combined with run-time observations of the system, enables the construction of more powerful services than those exported by the base system.

In this paper, we explore the application of gray-box techniques to the file placement problem. Specifically, to retain the convenience of the file system while regaining control over placement, we introduce PLACE (Positional LAyout ControllEr), a system that exploits gray-box techniques to give applications improved control over file placement. The system is depicted in Figure 1.

The most important component of PLACE is the PLACE Information and Control Layer (ICL). The PLACE ICL allows applications to group files or directories into localized portions of the disk, specifically into a particular group. Proper placement of data can improve both read and write performance; by collocating files that are likely to be accessed at nearly the same time, applications can improve their performance by "short-stroking" the disk, *i.e.*, reducing the cost of seeks by limiting arm movement to a certain portion of the disk. Applications that do not use the PLACE library operate as expected.

The key to the PLACE implementation is the *shadow directory tree* (SDT). The SDT is a hidden control structure that the PLACE ICL uses to control where files are placed on disk. By carefully creating this structure and exploiting our gray-box knowledge of file system behavior, the SDT enables the PLACE ICL to place files according to user preferences in a correct and efficient manner. Creating and maintaining this structure is one of the central challenges in implementing PLACE.

We first evaluate the PLACE ICL with a set of microbenchmarks to understand the basic costs and potential benefits of using PLACE. We find that the costs of using PLACE are reasonable, although a controlled file or directory creation is more costly than standard versions

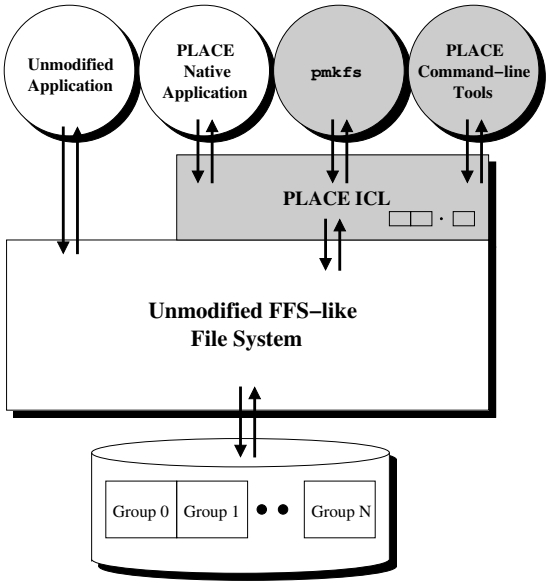

Figure 1: **The PLACE System.** *The PLACE system consists of three components, highlighted in gray in the figure. The most important component is the PLACE Information and Control Layer (ICL), which uses gray-box techniques to discover information about the file system, and then exploits that knowledge to enable applications that link with it to control file and directory layout. The two other components of PLACE are the tool* `pmkfs`*, which is used to initialize the PLACE on-disk structures, and a set of PLACE command-line tools. PLACE currently works on top of "FFS-like" file systems by learning of their internal group structure and exposing this structure through the PLACE ICL.*

of either operation in our prototype implementation. We also find that the potential benefits are substantial; random I/O performance improves dramatically when related data items are grouped into a small portion of the disk, and large files can be placed onto the outer tracks of a disk to improve throughput due to zoning effects [15].

We then demonstrate the utility of PLACE with two separate application studies. In the first, we show how a web server can use PLACE to group files that exhibit temporal access locality. Even when utilizing simple placement heuristics, collocation with PLACE improves web server throughput noticeably. In the second, we show how a high-speed user-level benchmarking tool called FAST can use PLACE to rapidly construct its testing infrastructure. Through controlled placement of files, FAST can extract important disk characteristics such as seek time and bandwidth in seconds.

The rest of this paper is organized as follows. In Section 2, we describe the design and implementation of PLACE, and in Section 3, we measure its costs and show its benefits. In Section 4, we present two case studies of PLACE usage. We describe related work in Section 5, future directions in Section 6, and conclude in Section 7.

2 PLACE: Design and Implementation

In this section, we describe the PLACE system for controlling file layout. We first provide background, describe our goals in implementing PLACE, and then describe the API as exposed through the PLACE ICL. After presenting the programming interface, we discuss the PLACE implementation, including system initialization and the shadow structures it uses to control file placement. We also discuss general operation, issues of concurrency, and some limitations of the current implementation.

2.1 Background

Many modern UNIX file systems are based on the Berkeley Fast File System[13], including direct descendants found in the BSD and Solaris families, and intellectual descendants such as Linux ext2 [29]. One of the main innovations of FFS is the emphasis placed upon *locality* – by placing related data objects near one another on disk, FFS provided a quantum leap in performance over file systems that scattered data across the disk in an oblivious manner.

The primary construct used in FFS to manage disk locality is the *cylinder group* (or *block group* in ext2, terms that we will use interchangeably for simplicity). A cylinder group divides the disk into a number of contiguous regions, each of which consists of inodes, data blocks, bitmaps for tracking inode and block usage, and a small number of blocks that store implementation-specific information. By placing related data objects into a cylinder group, and conversely spreading unrelated objects across different groups, locality of access can be achieved.

The difficulty, of course, is deciding exactly which objects are "related" and which are not. Typically, simple heuristics based on the file system namespace are used. Specifically, to group related objects, most implementations place the inodes and data blocks of files within the same directory into the same group, assuming locality of access among those files. Conversely, new directories are placed in different groups, so as to spread presumably unrelated files across the disk (thus leaving some "room to grow" in each group). The original FFS implementation (and some descendants) spread large files across groups, so as to avoid filling one group with a single large file.

In designing the PLACE ICL, we seek to exploit our gray-box knowledge of how FFS-like systems perform file layout in order to allow users to better control where their files are placed on disk. We also wish to understand the limits of such gray-box control, including the types of functionality that cannot be realized on top of modern file systems.

2.2 Design Goals

In designing PLACE, we have the following goals:

- **Simple and intuitive control over layout:** Applications should be given a straightforward representation of disk locality, which they can then exploit with their own application-specific knowledge to improve I/O performance.
- **Easy to use:** PLACE should be as easy to use as possible – no substantial code modifications should be required. Both programming APIs and command-line tools should be provided.
- **Compatible with non-PLACE applications:** Applications that do *not* use PLACE on top of a given file system should operate as before, *i.e.*, basic file system structure and usage for unmodified applications should not change.
- **Unaffected file system namespace:** Applications (and users) should be able to name files according to whatever conventions they desire – layout should not be dependent upon specialized naming schemes.

As we will see below, these goals impact both the design and the implementation of PLACE.

2.3 Abstractions and API

As its basic abstraction, PLACE exposes the underlying *groups* of FFS-like file systems to applications that link with the PLACE library. Applications can then use knowledge of their own access patterns to place related files and directories into a specific group, thus exploiting spatial locality for improved performance.

The number of each group also provides applications with two other pieces of information. First, applications can safely assume that files in proximate groups are reasonably "close" to one another, *e.g.*, an object in group 1 is close to an object in group 2, but not likely to be very close to an object in group 55. Second, lower group numbers are located near the outer tracks of the disk, whereas higher group numbers are located near the inner tracks. Applications may wish to utilize zone-sensitive placement for large files and thus improve throughput.

Note that more abstract "virtual" groupings and even group hierarchies could be layered on top of the physical group interface if desired. However, for simplicity, we focus solely on this lowest level of abstraction in the rest of this paper.

To allow applications to place files and directories into specific groups, PLACE provides two basic functions to applications:

- `Place_CreateFile(char *pathname, mode_t mode, int group);` Creates a file specified by `pathname` and with mode set to `mode` in group number `group`. The first two arguments are identical to the `creat()` system call.
- `Place_CreateDir(char *pathname, mode_t mode, int group);` Creates a directory specified by `pathname` with mode set to `mode` in group number `group`. The first two arguments are identical to the `mkdir()` system call.

The `Place_CreateFile` call allows the fine-grained placement of files into particular groups, whereas the `Place_CreateDir` function allows applications to create a directory in a controlled manner. Subsequent file allocations in that directory (through PLACE or not) are then likely to be collocated, due to standard FFS policy.

Of course, PLACE may not be able to allocate the file or directory into a particular group, due to insufficient resources (*i.e.*, there are no free data blocks or inodes left in the group). In such a case, the standard behavior is for the routine to return an error indicating why and the object is not created. An alternative interface can be used in which the routines can instead search for a "nearby" group upon failure, and place the file or directory therein.

Several other utility and convenience functions are also provided. For example, applications can discover the number of groups in a given file system, the current utilization level of each group, and the number of the group that is currently the least utilized.

When a user does not wish to or cannot re-write an application to use the PLACE API, a set of command-line tools can be utilized instead. These tools allow users to move directories and files to specific groups, or to create them in specific groups; subsequent data access by unmodified applications will thus enjoy the benefits of the rearrangement.

2.4 Basic Operation

PLACE exploits the FFS tendency to use the file namespace as a hint for placement to gain control over file layout. To do so, PLACE must first create a structure in which new files and directories can be created in a controlled fashion; once created therein, the PLACE library then renames the files, moving them back into their proper location within the file system namespace. This file system structure, known as the *shadow directory tree* (SDT), is central to the PLACE implementation.

At initialization (a process that is performed once per new file system), PLACE produces an SDT structure that appears in the file system namespace as shown in Figure 2.

There are three important entities found within the SDT. First, the `.superblock` file contains persistent in-

```
/.hidden/.superblock
/.hidden/.concurrency
/.hidden/D1/
/.hidden/D2/
...
/.hidden/Dn/
```

Figure 2: **The Shadow Directory Tree.** *The hidden shadow directory tree structure is presented. The* `.superblock` *file contains persistent information needed by PLACE, and the* `.concurrency` *file is used to manage multi-user access. Finally, the directories* `D1` *through* `Dn` *are used to control file placement.*

```
repeat
    tmp = PickNewName();
    mkdir(tmp);
    if (InDesiredGroup(tmp)) then
        break;
    end
    FillOtherGroups() ;
until forever;
rename(tmp, dirname);
```

Algorithm 1: **Directory Creation Algorithm.** *The basic algorithm used to create a directory in a specific group on disk is presented.*

formation about PLACE. Second, the `.concurrency` file is used to manage concurrent access to files through the PLACE API. Both of these files are discussed in more detail below. Third, and most interesting, is the set of directories named `D1` through `Dn`, where `n` is the number of cylinder groups in the file system. The initialization procedure (also described in more detail below) ensures that directory `Dk` is placed into cylinder group `k`. Note that all of these structures are placed in a "hidden" directory so that most applications will not see them when traversing the directory tree.

2.4.1 Controlling File Creation

With the SDT in place, creating a file in a particular group is straightforward. An application calls `Place_CreateFile`, passing in the pathname of the file to be created, the mode bits, and the desired group `k` within which to place the file. Internally, the PLACE ICL creates a file in the `Dk` shadow directory, and then simply calls `rename()` to put the file in the proper location in the namespace.

PLACE also checks to make sure that the file is allocated to the group the user requested, by looking up the i-number of the newly allocated file. During initialization (described below), PLACE learns of and records the i-number to group mapping, and uses that information here to determine if the allocation was successful.

2.4.2 Controlling Directory Creation

Placing a directory into the proper group with `Place_CreateDir` is more challenging; creating a directory in the proper `Dk` shadow directory does not suffice, as FFS-like file systems will place the child directory in a different cylinder group than its parent. Thus, a different approach is required, as shown in Algorithm 1.

The algorithm works by creating a temporary directory, checking if it is in the desired group (via its i-number), and repeating this process until the temporary directory is created in the correct group. When that directory is created, it is renamed to the proper location in the namespace.

One complication arises due to the particular directory allocation policies of some FFS-like file systems. For example, the Linux ext2 policy searches for a group with an above-average number of free inodes and the fewest allocated data blocks, whereas NetBSD FFS picks the group with an above-average number of free inodes and the fewest allocated directories. Thus, the algorithm must be willing to create temporary files as well as directories to coerce the file system into creating a directory in the desired group. This process, referred to in Algorithm 1 as FillOtherGroups(), creates some number of files in each of the non-target groups. To ensure that the files are not spread across different groups in an uncontrolled manner, PLACE creates "small" files (*i.e.*, files that do not utilize any indirect pointers).

Unfortunately, this basic algorithm can be quite slow, as we will demonstrate in Section 3. To speed up the process in the common case, we build a *shadow cache* of directories with known group numbers within the SDT. Before attempting to create a new directory within a particular group, the directory creation algorithm first consults the shadow cache to see if a directory within that group already exists; if so, PLACE simply renames that directory and is finished, thus avoiding the expensive directory creation algorithm.

If PLACE does not find the appropriate directory in the cache, it performs the full-fledged algorithm as described above. In this case, the directories that are created during the algorithm can be added to the cache, thus repopulating the shadow cache periodically.

2.5 SDT Initialization

We now discuss the initialization process required by PLACE, as encapsulated within a tool we call `pmkfs` (for "PLACE mkfs"). There are two steps to `pmkfs`. First,

`pmkfs` discovers various system parameters which are used in the algorithms described above. Second, `pmkfs` creates the SDT on-disk data structures and populates the shadow cache.

2.5.1 Parameter Discovery

PLACE requires several pieces of information to create the on-disk structures to support controlled allocation. These are the number of groups in the file system (N_{grp}), and the number of blocks (B_{grp}) and inodes (I_{grp}) per group. The total number of blocks and inodes in the system are readily available via the `statfs()` system call.

Finding the number of groups is slightly more challenging. Our current algorithm calculates this number by allocating directories and recording the difference in the inode numbers of subsequently allocated directories. Since each directory is likely to be in a new group, the most common difference is the number of inodes per group. PLACE detects when allocation has "wrapped around" by the fact that a new directory will have an i-number that is quite close to a previously allocated directory (usually, one more). Once one knows I_{grp}, one can calculate the group number (G_{num}) of an object from its inode number (I_{num}) by computing: $G_{num} = (I_{num} - 1)/I_{grp}$.

The system also calculates the number of direct pointers used in an inode (*i.e.*, the size of a "small" file), which is required for the directory creation algorithm to work across multiple FFS platforms. This value is discovered by synchronously writing blocks into a file, and monitoring the number of free blocks in the file system via `statfs`. The "small" file size is discovered at the point where a single allocating block write decreases the free block count by two blocks, indicating that an indirect block has been allocated.

In the current implementation, PLACE requires exclusive access to an empty file system during initialization. The only reason for this restriction is that the value of I_{grp} is not exported by the file system and the procedure described previously must be used to determine it. If this number were made available from an outside source (*e.g.*, the system administrator), then PLACE could be initialized on top of a system already in use.

2.5.2 SDT Creation

In the second step, `pmkfs` stores the necessary information into the `.superblock` file, and then creates the directory tree containing directories `D1` through `Dn`, assuming `n` groups. The process of creating these directories is identical to the directory creation algorithm found in Algorithm 1. As in the typical directory creation procedure, excess directories that are created are added to the shadow cache. In general, PLACE tries to maintain a minimal threshold of shadow directories per group, so as to avoid the costly directory creation algorithm.

To obtain a better understanding of what this threshold should be, we examined file system traces from HP Labs [17]. During a typical busy day, we found that a few thousand long-lived directories were created, giving us a rough upper bound on the number of shadow directories PLACE would need to maintain to absorb a day's worth of controlled directory creation in that environment.

2.6 Other Issues: Crash Recovery and Concurrency

During both file and directory creation, PLACE may create files and directories in the SDT, and thus there is the potential that data will accrue there over time; this will occur, for example, when a file is created in the SDT but the system crashes before the `rename` has taken place, or worse, if a job is killed in the midst of a PLACE library call. PLACE must thus include a basic crash recovery mechanism in order to periodically remove these files. We refer to this process as *SDT cleaning*.

Our current implementation of the SDT cleaner scans the SDT directory structures and removes any data objects that are "old" and thus left over from system crashes. As for how often to run the cleaner, many alternatives are possible. Our current implementation invokes the cleaner once every c invocations of PLACE (currently, c is set to 1000, which is probably too conservative), and whenever the longer directory-allocation process is run. Other alternatives include running the cleaner once per time interval (*e.g.*, once every day), or in a background process.

New issues also arise when considering PLACE usage under multiple processes or users. Concurrent use of PLACE by different processes is only a problem in the current implementation when using the basic algorithm to allocate a directory. In that situation, competing controlled directory creations in different groups could lead to significant difficulty in creating a directory in the desired location(s). To avoid this problem, PLACE acquires an advisory lock on the `.concurrency` file during this mode. This lock is only used to signify usage of the basic algorithm. In practice the usage of the basic algorithm repopulates the shadow cache, reducing the need for this mode of operation. A more cooperative approach is possible, where processes share the work of gaining control, but this would introduce significant complexity.

Multiple users also introduce a new issue, particularly as to whether the SDT should be shared or private per user. Sharing requires some level of trust among applications, as the SDT must be in a writable location. Thus, a shared SDT is vulnerable to many types of attacks (*e.g.*, changing the structures of PLACE to lead to poor allocations, or filling the SDT and causing a denial of service).

In many environments, this is not a problem, as a single user or application may have sole access to the file system. However, in less trustworthy settings, the SDT could be replicated on a per-user basis; although this increases space utilization and duplicates effort, it circumvents the security issues that arise due to sharing.

2.7 Limitations

The primary limitation of PLACE is that it is currently implemented only for FFS-like file systems. However, most modern UNIX file systems are FFS-like, and recent features, including journaling [29] within ext3 or the Soft Updates found within the BSD family of FFS implementations [23], do not affect our ability to control file placement with the techniques described above.

Another limitation arises due to the internal implementation of some FFS implementations, which spread larger files across different cylinder groups in order to avoid filling a single group too quickly [13]. This FFS behavior prevents PLACE from controlling where large files are laid out on disk, and thus we provide an interface to query PLACE as to the largest file size whose allocation can be "guaranteed" to be controllable. One notable exception to this standard FFS implementation strategy occurs within ext2, which does not spread larger files across different groups; this implementation strategy hints at what gray-box implementors would like to find inside of the systems they build on top of – behavior that is simple to understand and thus relatively easy to control.

PLACE also does not directly allow for fine-grained placement of files *within* a particular group. However, applications can modify the order of file creation to pack files into a group in a controlled fashion [1].

One alternative that we had initially explored overcomes these limitations but does not mesh well with applications that do not use PLACE. In this alternative approach, PLACE initially fills the target file system with a set of dummy files; by discovering the exact locations of each file, PLACE can then free up space whenever applications request new space, and thus *all* data allocations can be controlled. However, we deemed this approach unacceptable, as unmodified applications would not work correctly – to those applications, the file system appears as if it is full.

3 Analysis

In this section, we analyze the behavior of PLACE, demonstrating its functionality and its basic overheads. We first discuss the experimental environment, and then proceed through a series of microbenchmarks, demonstrating the effectiveness of layout control, and revealing the costs of system creation and usage. We then show how much improvement can be expected when reorganizing data and controlling layout to account for zoned-bit recording. Finally, we discuss our experience upon a broader range of OS platforms.

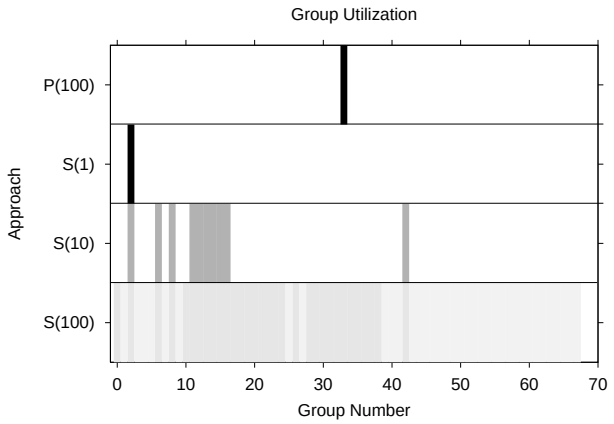

Figure 3: **Controlled Allocation.** *The graph depicts four different experiments, each of which creates 250 200-KB files. In the first three, the standard file system interfaces are used, but the number of directories under which the files are created is varied, from 1 to 10 to 100; these three experiments are labeled S(1), S(10), and S(100), respectively. In the fourth experiment, the PLACE API is used to create those files under 100 directories, but to place them in a single group in the middle of the disk (labeled P(100) in the graph). The group number is varied along the x-axis, and the shaded bar indicates some data has been placed in a particular group, with darker bars indicating more data. The* `debugfs` *command is used to gather the needed information.*

3.1 Experimental Environment

We present results with PLACE on top of the Linux 2.2 ext2 file system. All experiments on this platform are performed on a 550 MHz Pentium-III processor, 1 GB of main memory, and a 9 GB IBM 9LZX disk. The default ext2 file system built over this disk consists of 68 block groups. We also report on our experience with other file systems at the end of the section.

3.2 Layout Control

We begin with a simple experiment to demonstrate that PLACE can effectively collocate files into a specific group on the disk. Specifically, we compare four different methods of creating a 50 MB directory tree, allocated across 250 uniformly-sized files. In the first three, we use the standard file system interfaces, and alter the number of directories under which to place the files, from 1 to 10 to 100. In the fourth, we use the PLACE ICL to create the files underneath of 100 directories, but direct the system

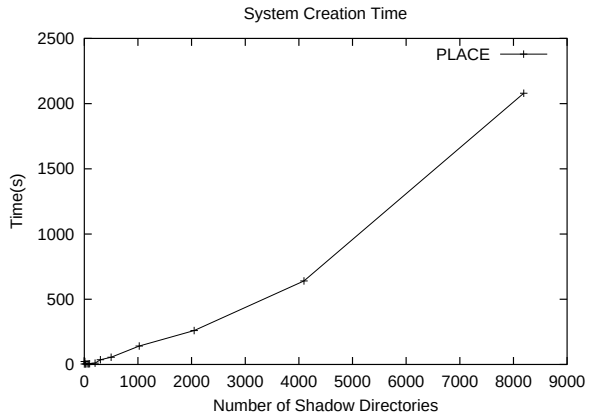

Figure 4: **System Initialization.** *System initialization time is plotted. Along the x-axis, we vary the number of shadow directories created per group, and the y-axis plots the total time for the initialization to complete.*

	Time (ms and Percentage)			
	0 B	*8 KB*	*64 KB*	*1 MB*
Base	0.12 7.7%	0.20 12.0%	0.79 34.5%	9.80 85.6%
State	1.19 75.5%	1.20 70.8%	1.20 52.3%	1.23 10.7%
Alloc	0.14 9.2%	0.15 8.7%	0.15 6.4%	0.15 1.3%
Ren	0.04 2.6%	0.04 2.5%	0.04 1.9%	0.08 0.7%
Misc	0.08 5.0%	0.11 6.4%	0.11 4.9%	0.20 1.7%
Total	1.57 ms	1.69 ms	2.29 ms	11.45 ms

Table 1: **File Allocation Overheads.** *Each result shows the average of 100 controlled file creations using the PLACE ICL. There was little variance (less than 0.04 ms) across the runs.*

	Time (ms and Percentage)			
	Shadow Cache	*Without Shadow Cache*		
		Min	Median	Max
Base	0.08 4.4%	0.10 3.0%	0.10 0.4%	0.00* 0.0%
State	1.17 63.4%	1.47 46.4%	1.19 4.8%	0.00* 0.0%
Alloc	0.24 12.7%	0.99 31.4%	5.46 22.1%	3.29 69.9%
Ren	0.04 2.0%	0.03 1.0%	0.03 0.1%	0.00* 0.0%
Clean	N/A	0.22 7.2%	17.5 70.7%	1.41 30.0%
Misc	0.32 17.5%	0.35 11.0%	0.47 1.9%	0.01 0.1%
Total	1.85 ms	3.16 ms	24.7 ms	4.71 s

Table 2: **Directory Allocation Overheads.** *Each result shows the average of 100 controlled directory creations using the PLACE ICL. Note that while most times are in milliseconds, the rightmost column (Max) shows time in seconds. The '*' indicates that the time shown is not actually zero, but appears as such due to rounding.*

to place the files and directories into a single group in the middle of the disk. Figure 3 shows the group utilization of each approach for directory trees of 50 MB of data.

As we can see from the figure, with more directories and the standard layout algorithms, the data from the files is scattered across the disk. In contrast, with PLACE, all of the data is located in the middle group of the file system, exactly as desired.

3.3 System Creation

Now that we have demonstrated basic control over layout, we seek to understand the costs of using the system. The first cost that we present is that of system initialization, as performed by the `pmkfs` tool. Figure 4 presents system initialization time.

The dominant cost of system initialization is in the number of shadow directories that are created within the shadow cache. Therefore, we present the sensitivity of initialization time to the number of shadow directories created per group. As one can see from the figure, this cost does not scale well with an increasing number of directories in the Linux ext2 system, as an increasing amount of data needs to be created in order to allocate directories across all of the groups successfully.

3.4 API Overheads

We next present the overheads of controlled file and directory creation via PLACE. Our goal here is to understand the costs of gray-box control over data placement. Table 1 breaks down the cost of creating different-sized files through the `Place_CreateFile` interface.

The costs presented in the table are broken down into five different categories, across four different file creation tests. The five categories are as follows: **Base**, the time to create the file itself through standard interfaces; **State**, the time to read the `.superblock` file to access system statistics and configuration information; **Alloc**, the time to control allocation (in this case, a *stat* system call to check the inode number); **Ren**, the time to rename the file into the correct namespace; and **Misc**, additional software processing overhead.

As we can see from the table, the PLACE API for file creation adds roughly a 1 ms overhead to file creation. This cost is mostly due to PLACE initialization, which would be amortized over multiple calls to the PLACE library. However, there is still significant overhead in the allocation, rename, and other software overheads. Finally, as file size increases, the overheads are also (unsurprisingly) amortized.

We next explore the overheads of directory creation via the `Place_CreateDir` API. Table 2 presents the cost breakdown of a controlled directory allocation, both with and without the shadow cache. Note that a new category is also included, labeled **Cleanup**, which includes time spent cleaning up the SDT after the directory-allocation process has run. Also note that **Alloc** in this case refers to

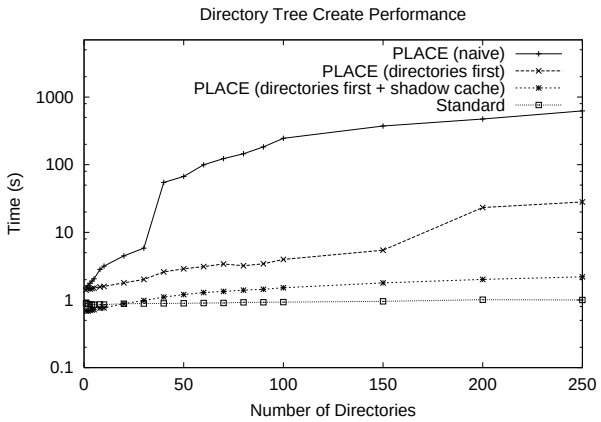

Figure 5: **Create Performance.** *The cost of moving a directory tree into a specific group is presented, varying the number of sub-directories in the structure along the x-axis, given a fixed amount of data (50 MB, spread evenly across 250 files). Four different approaches to creating the structure are compared, as described in the text. The y-axis presents the total time for the bulk collocation, on a log scale.*

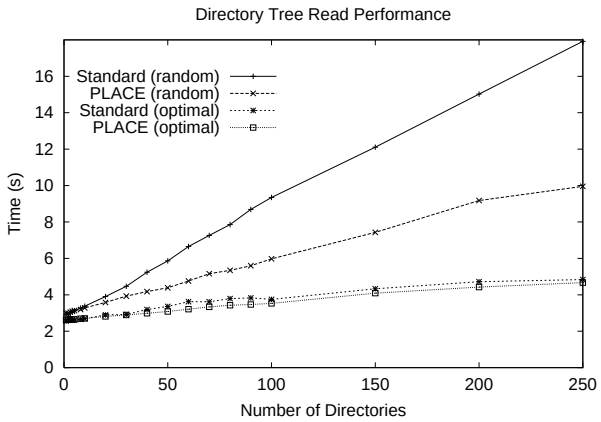

Figure 6: **Small-file Reads.** *The time to read 250 200-KB files (50 MB) of data is shown, under four different settings, varying the number of directories across which the data is placed. In two settings, the standard file system APIs are used to create the files. In the other two settings, PLACE is used to collocate all data into a single group. Two orders are shown: 'random', which reads the files in random order, and 'optimal', which reads them in a single scan of the disk.*

the costs of creating any necessary files or directories as required by the directory-allocation algorithm.

From the table, we can make a number of observations. First, with the shadow cache, the time for a controlled directory creation is reasonable, at roughly 1.85 ms (however, this value is still substantially higher than the base directory creation cost, which is approximately a factor of 20 faster). Second, without the shadow cache, times are higher, with a median cost of around 25 ms. The column that lists the maximum time without the shadow cache indicates the potential cost of running the full directory-creation process; in the worst case, it takes over 4.7 $seconds$ to create a directory in the correct group. This difficulty arises with a controlled creation within the last (and hence smaller) group; because ext2 allocates directories based on free bytes remaining, it takes an excessively long time to fill up the other groups and hence coerce a directory allocation into this last group. The **Base**, **Alloc**, and **State** times are essentially constant, and constitute a negligible part of the total in the maximum case. Since the shadow cache does not use the basic algorithm, it does not need to **Clean** afterward.

3.5 Bulk Collocation Costs

A common usage of PLACE is to move an entire directory tree into a specific group on the disk, which can be accomplished with one of the PLACE command-line tools. Thus, we were interested in what strategy this tool should take in moving a large amount of data from the source to its final destination within one group (or a small number of groups).

Figure 5 presents the time to perform this "bulk collocation" of 50 MB of data, again spread evenly across 250 200-KB files, under a varying number of sub-directories. Four schemes are compared. The first uses PLACE in a naive fashion, by creating directories and files in the target group recursively, and assuming that no shadow cache exists. This approach is dramatically slow, as the directory creation algorithm finds it increasingly difficult to force data into the target group. The second approach creates directories first, and performance improves tremendously, because the ext2 allocation policy uses the number of bytes allocated in its group-selection policy. Thus, by not creating files in the target group, it is much easier to coerce the system into choosing it. The third scheme shows the time for the second approach assuming that directories can be allocated from the shadow cache, which also improves the performance of the bulk collocation down to just a few seconds. Finally, a traditional directory-tree copy is shown as a comparison point; it is fast because it does not have any overhead associated with it, even though it is likely to spread data across the disk in a less localized manner.

3.6 Benefits of Collocation

To quantify the potential read performance improvement of PLACE, we perform a final set of microbenchmarks. Figure 6 shows the performance of the first set of tests, which present the time it takes to read a set of 250 200-KB files that have been collocated on the disk.

From the figure, we can see that if an application reads a

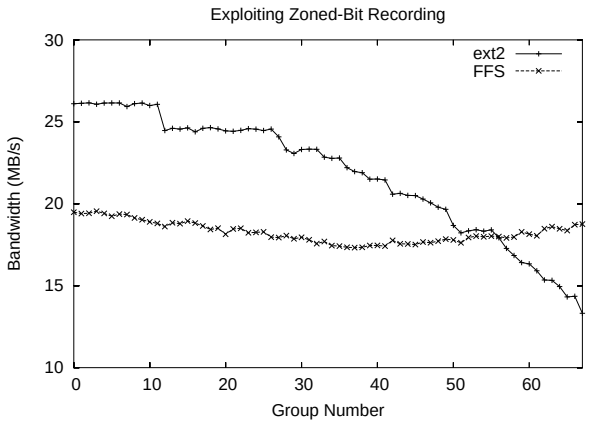

Figure 7: **Large-file Reads.** *The performance of reading a 100-MB file is shown on both ext2 and FFS, while varying the group in which the file is created along the x-axis. The file cache is flushed via a umount/mount cycle before the read to ensure that the disk bandwidth is properly measured. Each point is the average of three trials; the variance across all trials was low.*

set of files in random order, collocating them into a localized portion of the disk improves performance by almost a factor of two (the 'random' lines in the graph). However, if the files are read in the optimal order (essentially just scanning across the disk in a single sweep), the benefits of collocation are quite small; in this case, spreading data across the disk results in only a few additional seeks, and thus makes little overall difference in performance.

We also demonstrate how PLACE can be used to take advantage of the zoned bandwidth characteristics of modern disks [15]. Figure 7 plots the performance of large sequential file scans, when the files are placed into specific groups. The figure depicts the performance of PLACE on two file systems: standard ext2, and a modified ext2 which acts like traditional BSD FFS.

As one can see from the figure, on the ext2 platform, placing data in the lower-numbered groups corresponds directly to placing data onto the outer zones of the disk, thus improving performance. We also observe that when the same experiment is run on top of an "FFS" file system, the zoning nature of the disks are hidden; because FFS spreads blocks of large files across the disk, PLACE cannot control the placement of those blocks.

3.7 Experience with Other Systems

Our primary focus has been on the ext2 file system, as it is a modern implementation of FFS concepts and a popular file system in the Linux community. However, we designed many aspects of PLACE with more general FFS-like systems in mind; therefore, we were interested in studying the behavior of PLACE on other platforms.

Our first test of generality was to run PLACE on top of an ext3 file system, the journaling version of ext2 [29]. Because ext3 goes to great lengths to preserve backwards-compatibility with ext2, the same on-disk structures are utilized. Thus, we were not surprised to find that PLACE works without issue on top of ext3.

We also tested PLACE on top of an implementation of the FFS [13] allocation algorithms in the Linux kernel. PLACE worked without modification in this environment, with the limitations discussed in Section 2.7 and shown directly in Figure 7 relating to the placement of large files.

4 Case Studies

In this section, we describe two different example uses of the PLACE library. In the first, we demonstrate how a web server can reorganize files with PLACE so as to improve server throughput and response time. In the second, we describe how a high-speed file system benchmarking infrastructure can use PLACE to quickly extract I/O characteristics from the underlying system.

4.1 Improving Web Server Throughput

In our first example, we apply PLACE in order to understand the potential performance improvement in a web server environment. By reorganizing files such that the most popularly accessed files are close to one another on disk, seek costs can be reduced. Web service is a particularly good target for PLACE, as the structure of a typical web directory tree does not necessarily match the locality assumptions encoded into most FFS-like file systems. Further, there is no need to change the source code of the web server; the reorganization can be performed off-line via command-line tools.

We study the potential benefits through a simplified trace-based approach. We utilize a web trace from the University of Wisconsin-Madison web server. The trace is first preprocessed to remove requests that do not induce file system activity, such as errors and redirects, and the only requests that remain are ones that transfer data and HTTP 304 replies (a reply to a cache coherence check). The trace contains roughly 2.6 million requests, and accesses a total directory tree size of 720 MB.

To understand the potential gains of collocation, we run the trace through a file system request generator. For each trace entry, the request generator invokes the appropriate file system call and records the response time. Specifically, to model HTTP 304 requests, the generator performs a `stat` system call on the file, and for requests that transfer data, it maps the file into memory and touches every page. Although this approach does not capture the full complexity of a web environment, it should give us a

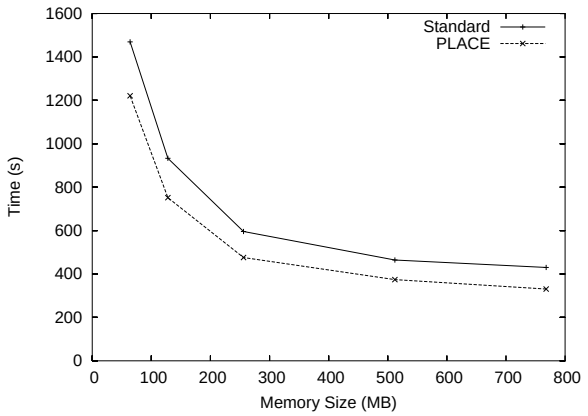

Figure 8: **Web Server Performance.** *The time to play back the I/O component of a web trace is shown. The standard line plots the performance with typical layout, and the PLACE approach packs all data into a small portion of the disk. Each point represents the mean of three trials (the variance was low), and the size of the directory tree being served is 720 MB.*

baseline for the potential performance improvement from file system reorganization.

We utilize the PLACE command-line tools to collocate the directory tree into the outer-most tracks of the disk, and compare this organization to a directory tree spread across the drive as determined by typical file system heuristics.

Figure 8 gives the time to replay the trace as a function of the amount of memory. Across the range of memory sizes, collocation with PLACE improves performance by roughly 20%. These benefits result directly from a reduction in seek costs, as demonstrated by further instrumentation of our testing apparatus. Specifically, by recording the group number of each file access, we can compute the average *group distance* traversed between requests. For the standard layout, the average number of groups between requests was 6.02, while for PLACE it was 0.52.

However, overall performance gains were limited due to the access patterns found in this particular web trace. In the trace, 76% of the requests were for 122 files in a single image directory. These files are thus collocated under standard FFS policy, reducing the need for PLACE-assisted file placement. However, even in such an environment, PLACE was able to improve performance in a simple and direct manner; a greater benefit can be expected in environments where access patterns do not match directory structure so closely.

4.2 Rapid File System Microbenchmarking

In our second example, we examine the use of PLACE in a different context, that of fast discovery of I/O performance characteristics. Many tools have been developed over time that extract performance characteristics from the underlying system [5, 14, 20, 21, 28]. However, many of these benchmarking tools need to be run as root, and all run for an uncontrolled (and potentially lengthy) amount of time. For example, Chen and Patterson's self-scaling benchmark runs for many hours (even days) before reporting results back to the user [5].

In some settings, it would be quite useful to have a system benchmarking tool that ran quickly, perhaps trading accuracy for a shorter run-time. For example, when running an application in a foreign computing environment (*e.g.*, Seti@Home [27], or in any wide-area shared computing system such as Condor [12] or Globus [8]), a mobile application needs to quickly extract the characteristics of the underlying system so that it can parameterize itself properly to the system. Further, the benchmark must be run entirely at user-level, requiring no special privileges to discover system parameters.

Thus, we develop the benchmarking tool FAST (Fast or Accurate System exTraction), that allows a mobile application to extract various performance characteristics from the underlying system under a fixed time budget and entirely at user level. Although FAST currently can extract information about both the I/O system and the memory system, only the I/O system component utilizes PLACE.

As an example of a mobile application, we examine the single processor version of NOW-Sort, a world-record-breaking sorting application [2]. While traditionally thought of in database contexts [16], sorting is also commonly found in many scientific computation pipelines [10], and therefore it is a reasonable candidate for mobile execution in scientific peer-to-peer shared computing systems [8].

NOW-Sort requires three parameters to tune itself to the host system. The first two are I/O parameters: the bandwidth expected from the local disk, and the worst-case seek time. With these two numbers, the sort can estimate how large its buffers must be during the merge phase in order to amortize seek costs. The third is the size of the caches in the memory-hierarchy. By sorting data in cache-sized chunks, sorting proceeds at a much faster rate [16].

The most difficult of these parameters to generally extract is the maximum seek cost. However, with the PLACE API, the FAST tool can create two files that are far apart on the disk, issue a synchronous update to the first, start a timer, issue a synchronous update to the second, and record the elapsed time of the second write, giving a coarse estimate of a full-stroke seek. Further refinements can be made over time, in order to remove rotational costs if so desired.

Table 3 presents the costs of running FAST on our test system. In this mode of operation, FAST runs as quickly as possible, garnering coarse estimates of the required system parameters. From the table, we observe that the

	Time (s)
Cache	0.73
Bandwidth	2.46
Max Seek	0.52
`pmkfs`	4.51
Total	8.22

Table 3: **FAST Performance.** *The table presents the time FAST takes to discover system parameters. In this mode, FAST is configured to run as quickly as possible, extracting coarse estimates but consuming less overall time.*

total time to extract the needed information for sorting is roughly 8 seconds. For sorts of massive data sets, spending the extra few seconds to configure the application is well worth the time. Finally, note that `pmkfs` is specialized to the task at hand; by giving it command-line options so as to prevent the creation of any shadow directories, initialization time is reduced to a small, fixed overhead.

5 Related Work

The work most directly related to PLACE is the gray-box File Layout Detector and Controller (FLDC) described in the original gray-box paper [1]. The FLDC has two components: the first can be used to decide in which order to access a set of files, and the second to re-write file within a particular directory so as to likely improve later accesses. Both components could be useful here; however, PLACE goes well beyond FLDC, exposing fine-grained control over file and directory layout to applications.

Applications have long sought better control over underlying operating system policies and mechanisms [26]. In response to this demand, previous research has developed new operating systems, including Spin [3], Exokernel [7], and VINO [22], that allow much-improved control over operating system behavior. The gray-box approach provides a different route to improved control over the underlying OS; by exploiting knowledge of OS behavior, PLACE demonstrates that file and directory layout can be realized at user-level.

The PLACE method of exposing group numbers is conceptually similar to the Exokernel philosophy of exposing physical names. PLACE treats the file system as the underlying entity and exposes its internal structure (*e.g.*, ext2 block groups), while Exokernel exposes the details of the hardware (*e.g.*, physical sector numbers).

Moving data blocks into a better spatial arrangement, as we do in the web server case study, has been explored in many other contexts. For example, in their work on disk shuffling, Ruemmler and Wilkes track frequency of block accesses, and reorder disk blocks to reduce seek times [19]. At a higher level, Staelin and Garcia-Molina rearrange where files are placed within the file system [25]. The major difference between these approaches and PLACE is that they are performed transparently to users and applications; no control is exposed. However, both are more sophisticated in tracking which blocks or files are accessed in temporal succession; we hope to develop an access-tracking tool in the future.

Our work on improving web server performance is similar to other work on improving web proxy performance. For example, Hummingbird is library-based file system designed for web proxies [24]. PLACE provides some of the same features of Hummingbird in that it allows the users to collocate files and does not tie locality to naming. In contrast, PLACE is implemented on top of an FFS-like file system, whereas Hummingbird performs its functions in a library that runs on a raw disk. Further, Hummingbird is specialized for a web-proxy environment, whereas PLACE is a general-purpose tool.

Finally, the FAST tool bears some similarity to recent work in database management systems. For example, in *online aggregation* [9], the DBMS returns an approximate result of a selection query to the user immediately, and includes a statistical estimate of the accuracy of the result. If the user allows the query to keep running, the system refines the result over time, and as more data is sampled, the answer becomes more precise. The FAST tool applies this same philosophy to a benchmarking system.

6 Future Work

A number of avenues exist for future research. First, we plan to explore the breadth of applicability of PLACE on top of other file systems. One platform we are interested in is the BSD family; we believe there are some new challenges in this domain, as more recent BSD implementations of FFS utilize the DirPrefs algorithm for directory group selection [6]. This algorithm places directories near their parents, in an attempt to increase the performance of certain common operations (*e.g.*, the unpacking of a large directory tree). Building a gray-box controller such as PLACE on top of DirPrefs would thus require that extra care be taken to spread directories across groups.

Building PLACE on top of a log-structured file system (LFS) [18] would also be interesting. For example, grouping of particular related files would generally be straightforward; if a user wishes to group two files, those files can be written out at the same time. However, other aspects make LFS more challenging, including grouping of files that span multiple segments and controlling the off-line behavior of the cleaner. More generally, we are interested in developing techniques that can be used to control allo-

cation across a broader range of file systems.

We would also like to investigate the utility of these methods on a range of different storage devices. Specifically, we would like to determine how useful such controlled file placement is on top of modern disk arrays.

Finally, a tool such as PLACE is a low-level mechanism for placing files in a controlled manner on disk; exactly which files should be placed together is a higher-level policy decision that requires detailed knowledge of how files are accessed over time. Thus, similar to previous work in data rearrangement [19], we plan to develop a tool to track how files and blocks are accessed and thus generate the necessary inputs for better file placement.

7 Conclusions

In the classic paper *Hints for Computer System Design* [11], Lampson tells us: "Don't hide power." Higher-level abstractions should be used to hide the *undesirable* properties; useful functionality, in contrast, should be exposed to the client. Many UNIX file systems do not expose explicit controls for laying out files according to user demands. Given standard layout heuristics, workloads that do not conform to the locality assumptions set in stone nearly 20 years ago perform poorly.

In this paper, we present the design, implementation, and evaluation of PLACE, a gray-box Information and Control Layer that exposes file and directory information to applications. By exploiting knowledge of internal algorithms that are common to FFS-like file systems, PLACE can control file and directory allocations.

Through microbenchmarks, we have shown that the costs of gray-box control are not overly burdensome, and that the potential benefits of controlled allocation are substantial. Through two case studies, we have demonstrated that the PLACE system can be used in realistic and diverse application settings. We have also discussed the limitations of PLACE as well as the gray-box approach to controlled allocation, highlighting the features of file system allocation policies that make it simple or difficult to build control on top of them.

The gray-box approach provides an alternative path for innovation. Instead of requiring changes to the underlying operating system, which may be difficult to implement, maintain, and distribute, a gray-box ICL embeds some knowledge of the underlying system, and exploits that knowledge to implement new functionality, often in a portable manner. One important question remains: what is the full range of functionality that can be implemented in the gray-box manner, and what are the ultimate limitations? With each ICL, we take another small step toward the final answer.

Acknowledgments

We would like to thank Nathan Burnett, Tim Denehy, Florentina Popovici, Vijayan Prabhakaran, and Muthian Sivathanu for their feedback on this paper. A special thanks to Muthian for his assistance with the HP traces. We also thank Geoffrey Kuenning for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions, which in combination have greatly improved the content of this paper. We thank John Heim and the staff of DoIt for providing a recent web trace that included path names, and Tom Engle for the implementation of the FFS allocation algorithm in the Linux kernel. Finally, we thank the Computer Systems Lab for providing a superb environment for computer science research.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, an IBM Faculty Award, and the Wisconsin Alumni Research Foundation.

References

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[2] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference on the Management of Data (SIGMOD '97)*, Tucson, Arizona, May 1997.

[3] B. N. Bershad, S. Savage, E. G. S. Przemyslaw Pardyak, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[4] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, CA, June 2002.

[5] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1992 ACM SIGMETRICS Conference*, pages 1–12, May 1993.

[6] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[7] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for

Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD International Conference on Management of Data (SIGMOD '97)*, pages 171–182, Tucson, Arizona, May 1997.

[10] K. Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, July 2001.

[11] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 33–48, Bretton Woods, NH, December 1983. ACM.

[12] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

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

[14] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Winter Technical Conference*, January 1996.

[15] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, January 1997.

[16] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. AlphaSort: A RISC Machine Sort. In *1994 ACM SIGMOD Conference*, May 1994.

[17] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14–29, Monterey, California, January 2002.

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

[19] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, Hewlett Packard Laboratories, Oct 1991.

[20] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

[21] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon, 1999.

[22] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[23] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, June 2000.

[24] E. Shriver, E. Gabber, L. Huang, and C. A. Stein. Storage Management for Web Proxies. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, pages 203–216, Boston, Massachussetts, June 2001.

[25] C. Staelin and H. Garcia-Molina. Smart Filesystems. In *Proceedings of the 1991 USENIX Winter Technical Conference*, Dallas, Texas, January 1991.

[26] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[27] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A New Major SETI Project based on Project Serendip Data and 100,000 Personal Computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.

[28] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[29] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.