A Gray-Box Approach To Controlling Data Layout: Techniques and Implementation

James 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-based file systems. PLACE exploits its knowledge of FFS layout policies in order to let users place files and directories into specific and localized portions of disk. Applications can use PLACE to colocate 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 they are not prohibitively burdensome, 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

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 [23]. Proper data allocation can exploit locality of access within a particular workload, increasing disk efficiency and thereby improving overall performance.

However, many file systems, while convenient to use, 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) [12] 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, and thus giving up convenience for control.

Gray-box techniques [1] 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; that knowledge plus observations of how the system is currently behaving allows 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, into a particular group. Proper placement of data can improve both read and write performance; by colocating files that are likely to be accessed at nearly the same time, applications can improve their perfor-

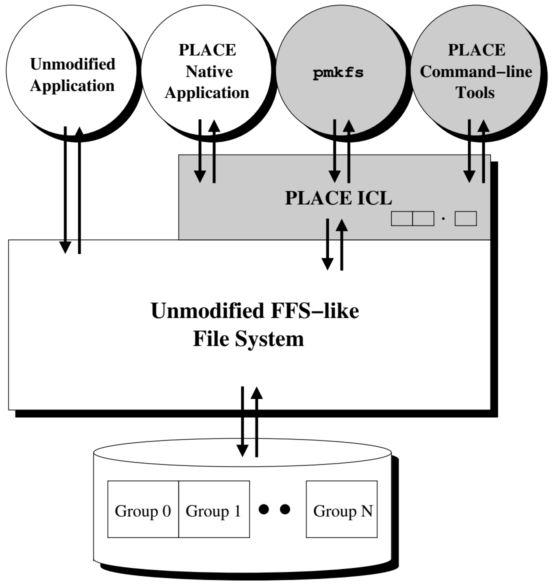

Figure 1: **The PLACE System.** *The PLACE system consists of three components, highlighted in gray in the figure. The most important component is the PLACE ICL, which uses gray-box techniques to discover information about the file system, and the exploits that knowledge to enable applications that link 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, learning of their internal cylinder group structure, and exposing this structure through the PLACE ICL.*

mance by "short-stroking" the disk (reducing the cost of seeks by limiting arm movement to a certain portion of the disk). Applications that do not use the PLACE library to access the file system are unaffected and operate as expected.

The key to the PLACE implementation is the *shadow directory tree* (SDT). The SDT is a hidden control structure which 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 thus one of the central challenges to an effective PLACE implementation.

We first evaluate the PLACE ICL with a set of microbenchmarks, to understand the basic costs and potential benefits of using PLACE. In general, we find that the costs of using PLACE are reasonable, although a controlled file or directory creation is still noticeably more costly than standard versions of either operation in our initial 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.

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, and thus improve overall throughput and reduce response time. In the second, we show how a high-speed user-level benchmarking system can use PLACE to rapidly construct its testing infrastructure.

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 basic overheads. In Section 4, we present two case studies of PLACE usage. We describe related work in Section 5, and conclude in Section 6.

2 PLACE: Design and Implementation

In this section, we describe the PLACE system for controlling file layout. We first provide some 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 the shadow structures it uses to control file placement, system initialization, 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[12], including direct descendants found in the BSD and Solaris families, and intellectual descendants such as Linux ext2 [26]. 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 consist of inodes, blocks, bitmaps to track inode and block usage, 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 in each group some "room to grow"). Some FFS implementations also 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-based 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 had the following goals:

- **Simple and intuitive control over layout:** Applications should be given a straight-forward 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 implementation of PLACE.

2.3 Abstractions and API

As its basic abstraction, PLACE exposes the underlying groups of FFS-based 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 locality. Group numbers also provide 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 colocated, due to standard FFS policy.

Of course, PLACE may not be able to allocate the file or directory into a particular group. In such a case, the standard behavior is for the routine to return an error and for the object not be 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, or ask PLACE to put an object into the currently least utilized group.

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 instead be utilized. These tools allow users to control the layout of directories and files; 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 in order 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, thus moving them back into their proper location within the file system namespace. This file system structure, known as the *shadow directory tree* (SDT), is thus central to the PLACE implementation.

At initialization (performed once per new file system), PLACE produces an SDT structure that appears in the file system namespace as follows:

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

There are three important entities found within the SDT. First, the `.superblock` file contains persistent info 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 straight-forward. 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 where 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. Upon failure, the file is not allocated and an error is returned (an alternative API places the file in a nearby group and informs the caller of the new group number).

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-based file systems will place the child directory in a different cylinder group than its parent. Thus, a more sophisticated approach is required, and is described as follows:

```
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 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 such a 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, Linux ext2 uses the number of bytes per group to place directories into the group with the lowest number of used bytes, whereas other FFS-based systems such as NetBSD FFS pick the target group by finding a group with many free inodes and the fewest allocated directories. Thus, the algorithm must also 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. In order to ensure that the files are not spread across different groups in the file system, 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 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 automatically 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` must discover various system parameters which are used in the algorithms described above. Second, `pmkfs` must create the SDT on-disk data structures and populate the shadow cache.

2.5.1 Parameter Discovery

PLACE requires several pieces of information in order to create the on-disk structures to support controlled allocation. These are the number of groups in the file system (N), and the number of blocks (B) and inodes (I) per group. The total number of blocks and inodes in the system is 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. Once one knows I, one can get the group number (GN) of an object from its inode number (IN) by computing: $GN = \frac{IN-1}{I}$.

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.

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 some minimal threshold of shadow directories per group, so as to avoid the costly directory creation algorithm.

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 directory structures and removes any data objects that are "old" and thus left-overs from system crashes. As for how often to run the cleaner, many alternatives are possible. Our current implementation invokes the cleaner once every n invocations PLACE (currently, n 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 (*i.e.*, 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 it is 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. To avoid this problem, PLACE acquires an advisory lock on the `.concurrency` file during this mode.

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 much of the work that needs to be done if many different users are running PLACE-enabled applications, 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 [26] within ext3 or Soft Updates that is found within the BSD family of FFS implementations [21], do not affect our ability to control file placement with the techniques described above.

It should also be noted that most of our experience has been with the Linux ext2 file system, although our algorithms are designed to work upon most FFS implementations with which we are familiar. The breadth of PLACE portability are discussed further in Section 3.

Controlling file placement from user-level in radically different file systems may be easier or more difficult depending on the exact system. For example, within a log-structured file system (LFS) [16], grouping of particular "related" files would generally be straight-forward; if a user wished to group two files, they could write out those files at the same time, and hence place them within the same log segment. However, other aspects make LFS more challenging, including the grouping of files that span multiple segments, and controlling the off-line behavior of the cleaner. Thus, achieving user-level control of a log-structured file system remains a potentially interesting area of future work.

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 [12]. This FFS behavior prevents PLACE from controlling where large files are laid out on disk, given its current implementation, 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.

One alternative that we had initially explored overcomes this limitation 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 could then free up space whenever applications requested new space, and thus *all* data allocations could be controlled. However, we deemed this approach unacceptable, as unmodified applications would not work correctly – to those applications, the file system appeared as if it was 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.

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. 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.

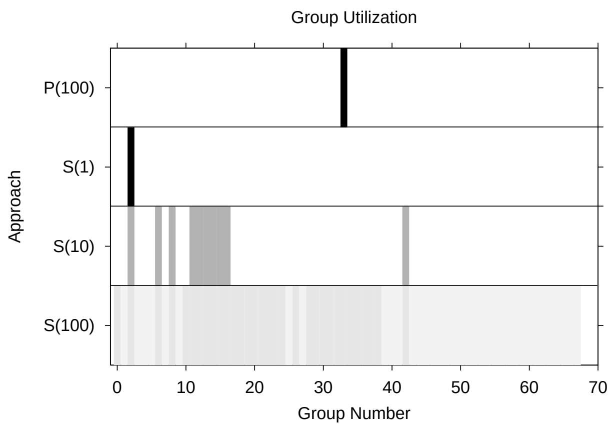

Figure 2: **Controlled Allocation.** *The graph depicts four different experiments, each of which creates 250 200-KB files. In the first three, the standard file system interfaces are used, but the number of directories under which the files are created is varied, from 1 to 10 to 100; these three experiments 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.*

3.2 Layout Control

We begin with a simple experiment to demonstrate that PLACE effectively can colocate files into a specific group on the disk. Specifically, we compare four different methods of creating 50 MB, 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 to place the files and directories into a single group in the middle of the disk. Figure 2 shows the group utilization of each approach organizations of 50 MB of data, in which we use the `debugfs` command to gather the needed information.

As we can see from the figure, with more direc-

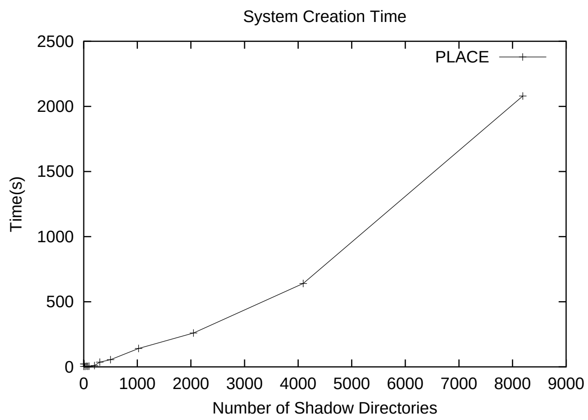

Figure 3: **System Initialization.** *System initialization time is plotted. The dominant cost of system initialization is the creation of shadow directories, and hence we vary the number created along the x-axis.*

tories and the standard layout algorithms, the data from the files gets scattered across the disk. In contrast, with PLACE, we can observe that 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 now 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 3 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. As one can see from the figure, this cost does not scale well with an increasing number of directories upon 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. A typical file system containing roughly 100 groups would create on the order of 20 shadow directories per group, and thus would take a few minutes to complete the procedure.

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.

	Time			
	0 B	*8 KB*	*64 KB*	*1 MB*
Base	7.7%	12.0%	34.5%	85.6%
State	75.5%	70.8%	52.3%	10.7%
Alloc	9.2%	8.7%	6.4%	1.3%
Rename	2.6%	2.5%	1.9%	0.7%
Misc	5.0%	6.4%	4.9%	1.7%
Total	1.57ms	1.69ms	2.29ms	11.45ms

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.*

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

	Time			
	Shadow Cache	*Without Shadow Cache*		
		Min	Median	Max
Base	4.4	3.0%	0.4%	0.0%
State	63.4	46.4%	4.8%	0.0%
Alloc	12.7	31.4%	22.1%	69.9%
Rename	2.0	1.0%	0.1%	0.0%
Cleanup	0.0	7.2%	70.7%	30.0%
Misc	17.5	11.0%	1.9%	0.1%
Total	$1.85ms$	$3.16ms$	$24.7ms$	$4.71s$

Table 2: **Directory Allocation Overheads.** *Each result shows the average of 100 controlled directory creations using the PLACE ICL.*

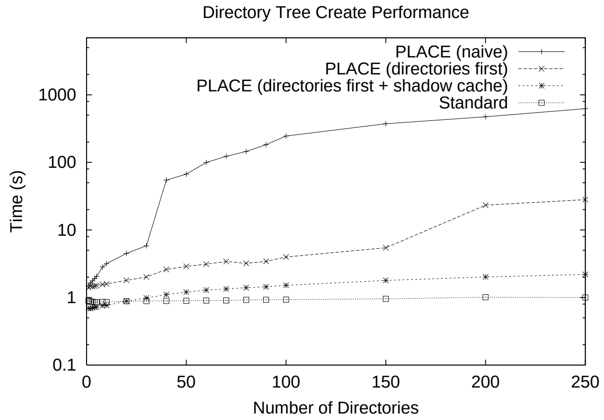

Figure 4: **Create Performance.** *The cost of moving a directory tree into a specific group is presented, varying the number of sub-directories in the structure along the x-axis, given a fixed amount of data (50 MB, spread evenly across 250 files). Four different approaches to creating the structure are compared, as described in the text. The y-axis presents the total time for the bulk colocation, on a log scale.*

From the table, we can make a number of observations. First, with the shadow cache, the time for a controlled directory creation is quite reasonable, coming in 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 unsurprisingly 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; it takes over 4.7 $seconds$ to finally create a directory in the correct group. In this case, the difficulty arises with a controlled creation within the last (and hence smaller) group; because ext2 allocates directories based on free bytes remaining, it takes a long time to fill up the other groups in order to coerce a directory allocation into this last group.

3.5 Bulk Colocation 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 4 presents the time to perform this "bulk colocation" 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 colocation 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 Colocation

To quantify the potential read performance improvement of PLACE, we perform a final set of microbenchmarks. Figure 5 shows the performance of the first set of tests, which present the time it takes

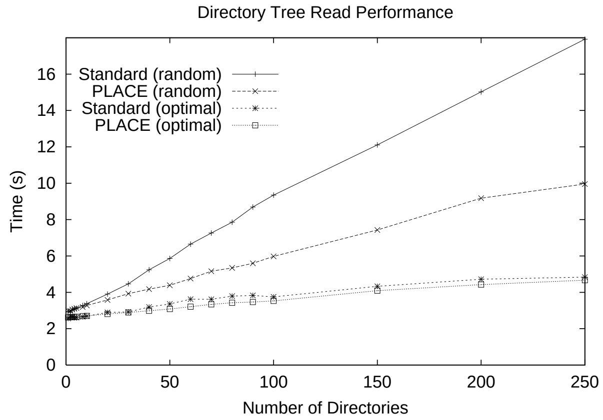

Figure 5: **Small-file Reads.** *The time to read 250 200-KB files (50 MB) of data is shown, under four different circumstances, varying the number of directories across which the data is placed. In two settings, the standard file system APIs are used to create the files initially, and thus the directories and files are spread across the groups of the file system. In the other two settings, PLACE is used to colocate all data into a single group. Two read orders are shown for both the standard approach and PLACE: 'random', which reads the files in random order, and 'optimal', which reads them in a single scan of the disk.*

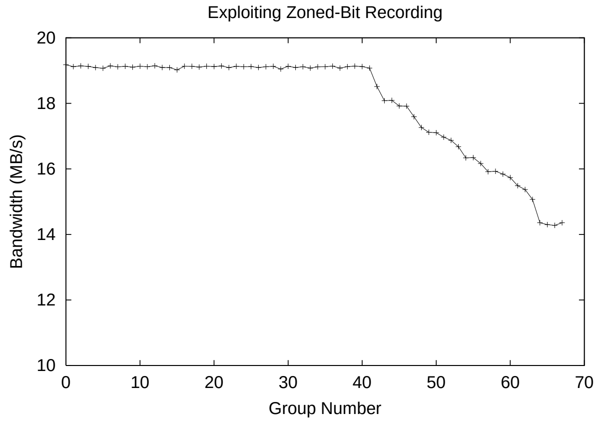

Figure 6: **Large-file Reads.** *The performance of reading a 100-MB file is shown, while varying the group in which the file is created along the x-axis. The file cache is flushed before the read to ensure that the disk bandwidth is properly measured. Each point is the average of three trials.*

to read a set of 250 200-KB files in that have been colocated on the disk.

From the figure, we can see that if an application reads a set of files in random order, colocating 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 colocation 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 [14]. Figure 6 plots the performance of large sequential file scans, when the files are placed into specific groups. As one can see from the figure, on the IBM disk under test, better zone placement improves performance by about 33%; other disks may exhibit even greater differentials across zoned regions. One limitation of this style of usage is that it is not generic to all FFS-like file systems; whereas the ext2 allocation strategy will keep data from a large file within a single cylinder, other FFS-based file systems likely will not.

3.7 Other Systems

Our primary focus has been upon 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 curious to find exactly how portable the PLACE algorithms for layout control were.

Our first test of generality was to run PLACE on top of an ext3 file system, the journaling version of ext2 [26]. 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 are currently investigating PLACE on top of other platforms as well, and plan to have this experience included in the final version of this paper if accepted. 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 [5]. This algorithm places directories near their parents, in an attempt to increase the performance of certain common operations (*e.g.*, the untar'ing 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.

4 Case Studies

In this section, we describe two different 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 a more traditional manner, 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 likely reduced. Web service is a particularly good target for PLACE, as the structure of a typical web directory tree does not match the locality assumptions encoded into most FFS-based file systems (*e.g.*, an images directory that contains images from all the pages scattered through the directory tree). 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 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 million requests, and a accesses a total directory tree size of 513 MB.

To understand the potential gains of colocation, we run the trace through a file system request generator. The generator reads in a trace entry, generates the appropriate file system call, and records the response time. Although this approach does not capture the full complexity of a web environment, it should give us a baseline for the potential performance improvement from file system reorganization.

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

In our initial performance tests, running the entire trace through on the standard file system organization takes roughly 406 seconds. On top of the PLACE-organized directory tree, the time improves to approximately 326 seconds, roughly a 20% decrease in total time. We are planning next to investigate tracing techniques that can give us better input as to which exact files to colocate for further improvements in performance.

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 [4, 13, 18, 19, 25]. 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.

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 [24], or in any wide-area shared computing system such as Condor [11] or Globus [7]), 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

	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.*

traditionally thought of in database contexts [15], sorting is also commonly found in many scientific computation pipelines [9], and therefore it is a reasonable candidate for mobile execution in scientific peer-to-peer shared computing systems [7].

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 [15].

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 figure, we observe that the 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 most directly related work 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 to decide in which order to access a set of files, and the second to re-write out files 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 the underlying operating system's policies and mechanisms [23]. In response to this demand, previous research has developed new operating systems, including Spin [3], Exokernel [6], and VINO [20], 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 perhaps be realized at user-level.

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 [17]. At a higher level, Staelin and Garcia-Mollina rearrange where files are placed within the file system [22]. 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, and thus are likely to be more successful in arriving at a better rearrangement.

Finally, the FAST tool bears some similarity to recent work in database management systems. For example, in *online aggregation* [8], 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 Conclusions

In the classic paper *Hints for Computer System Design* [10], 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 ICL known as PLACE that exposes file and directory layout control to applications. By exploiting knowledge of the internal algorithms that are common to FFS-like file system implementations, PLACE enables users to 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 can be implemented in the gray-box manner, and what are the ultimate limitations? With each ICL, we make another small step towards the final answer.

References

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Symposium on Operating Systems Principles*, pages 43–56, 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] 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.

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

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3), 1984.

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

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

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

[16] 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.

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

[18] 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.

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

[20] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI II*, 1996.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.