FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption

Hai Huang, Wanda Hung, and Kang G. Shin
Dept. of Computer Science and Electrical Engineering The University of Michigan
Ann Arbor, MI 48105, USA
haih@eecs.umich.edu, wanda@eecs.umich.edu, kgshin@eecs.umich.edu

ABSTRACT

Disk performance is increasingly limited by its head positioning latencies, i.e., seek time and rotational delay. To reduce the head positioning latencies, we propose a novel technique that *dynamically* places copies of data in file system's *free blocks* according to the disk access patterns observed at runtime. As one or more replicas can now be accessed in addition to their original data block, choosing the "nearest" replica that provides fastest access can significantly improve performance for disk I/O operations.

We implemented and evaluated a prototype based on the popular Ext2 file system. In our prototype, since the file system layout is modified only by using the free/unused disk space (hence the name *Free Space File System*, or FS^2), users are completely oblivious to how the file system layout is modified in the background; they will only notice performance improvements over time. For a wide range of workloads running under Linux, FS^2 is shown to reduce disk access time by 41–68% (as a result of a 37–78% shorter seek time and a 31–68% shorter rotational delay) making a 16–34% overall user-perceived performance improvement. The reduced disk access time also leads to a 40–71% energy savings per access.

Categories and Subject Descriptors

H.3.2 [**File Organization**]: Information Storage; H.3.4 [**Performance Evaluation**]: Systems and Software

General Terms

Management, Experimentation, Measurement, Performance

Keywords

Data replication, free disk space, dynamic file system, disk layout reorganization

1. INTRODUCTION

Various aspects of magnetic disk technology, such as recording density, cost and reliability, have been significantly improved over the years. However, due to the slowly-improving mechanical positioning components (i.e., arm assembly and rotating platters) of the disk, its performance is falling behind the rest of the system and has become a major bottleneck to overall system performance.

To reduce these mechanical delays, traditional file systems tend to place objects that are likely to be accessed together so that they are close to one another on the disk. For example, BSD UNIX Fast File System (FFS) [12] uses the concept of *cylinder group* (one or more physically-adjacent cylinders) to cluster related data blocks and their meta-data together. For some workloads, this can significantly improve both disk latency and throughput. Unfortunately, for many other workloads in which the disk access pattern deviates from that assumed by the file system, file system performance can significantly degrade. This problem occurs because the file system is unable to take into account of runtime access patterns when making disk layout decisions. This inability to use the *observed* disk access patterns for data placement decisions of the file system will lead to poor disk utilization.

1.1 Motivating Example

Using a simple example, we now show how traditional file systems can sometimes perform poorly even for common workloads. In this example, we monitored disk accesses during the execution of a CVS *update* command in a local CVS directory containing the Linux 2.6.7 kernel source tree. As Figure 1(a) shows, almost all accesses are concentrated in two narrow regions of the disk, corresponding to the locations at which the local CVS directory and the central CVS repository are placed by the file system on the disk. By observing file directory structures, the file system is shown to do well in clustering files that are statically-related. Unfortunately, when accesses to files from multiple regions are interleaved, the disk head has to move back and forth constantly between the different regions (as shown in Figure 1(b)), resulting in poor disk performance in spite of previously-known efforts made for reasonable placements. Consequently, users will experience longer delays and disks will consume more energy. In our experiment, the CVS update command took 33 seconds to complete on an Ext2 file system with an anticipatory I/O scheduler. By managing the disk layout dynamically, the update command took only 22 seconds to complete in our implementation of FS^2.

Traditional file systems perform well for most types of work-

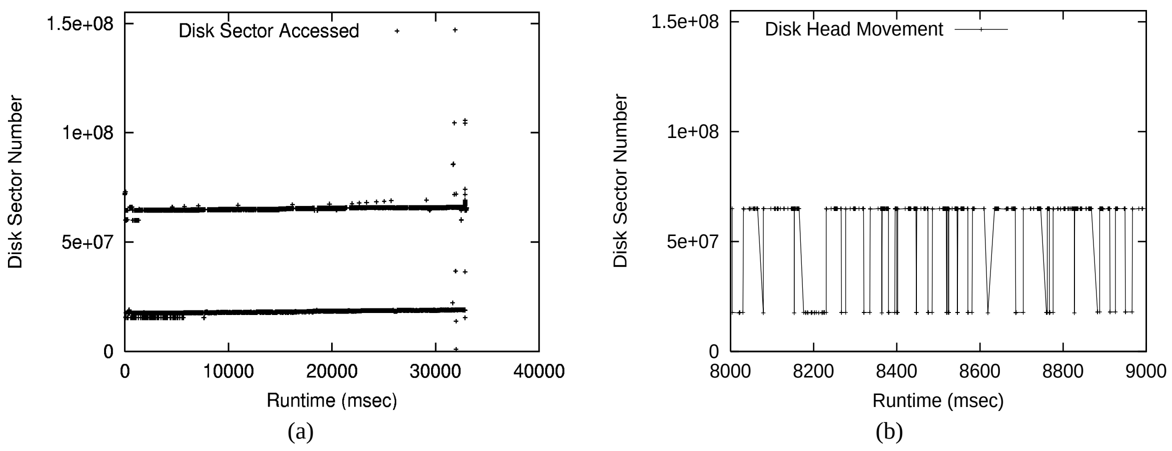

Figure 1: Part (a) shows disk sectors that were accessed when executing a *cvs -q update* command within a CVS local directory containing the Linux 2.6.7 source code. Part (b) shows the disk head movement within a 1-second window of the disk trace shown in part (a).

loads and not so well for others. We must, therefore, be able to first identify the types of workloads and environments that traditional file systems generally perform poorly (listed below), and then develop good performance-improving solutions for them.

Shared systems: File servers, mail servers, and web servers are all examples of a shared system, on which multiple users are allowed to work and consume system resources independently of each other. Problems arise when multiple users are concurrently accessing the storage system (whether it is a single disk or an array of disks). As far as the file system is concerned, files of one particular user are completely independent of those of other users, and therefore, are unlikely to be stored close to those of other users on disk. Thus, having concurrent disk accesses made by different users can potentially cause the storage system to become heavily multiplexed (therefore less efficiently utilized) between several statically-unrelated, and potentially-distant, disk regions. A PC is also a shared system with similar problems; its disk, instead of being shared among multiple users, is shared among concurrently-executing processes.

Database systems: In database systems, data and data indices are often stored as large files on top of existing file systems. However, as file systems are well-known to be inefficient in storing large files when they are not sequentially accessed (e.g., transaction processing), performance can be severely limited by the underlying file system. Therefore, for many commercial database systems [15, 22], they often manage disk layout themselves to achieve better performance. However, this increases complexity considerably in their implementation. For this reason, many database systems [31, 40, 15, 22] still rely on file systems for data storage and retrieval.

Systems using shared libraries: Shared libraries have been used extensively by most OSs as they can significantly reduce the size of executable binaries on disk and in memory [32]. As large applications are increasingly dependent on shared libraries, placing shared libraries closer to application images on disk can result in a smaller load time. However, since multiple applications can share the same set of libraries, determining where on disk to place these shared libraries can be problematic for existing file systems as positioning a library closer to one application may increase its distance to others that make use of it.

1.2 Dynamic Management of Disk Layout

Many components of an operating system—those responsible for networking, memory management, and CPU scheduling—will re-tune their internal policies/algorithms as system state changes so they can most efficiently utilize available resources. Yet, the file system, which is responsible for managing a slow hardware device, is often allowed to operate inefficiently for the entire life span of the system, relying only on very simple heuristics based on static information. We argue, like other parts of an OS, the file system should also have its own runtime component. This allows the file system to detect and compensate for any poorly-placed data blocks. To achieve this, various schemes [36, 44, 2, 16] have been proposed in the past. Unfortunately, there are several major drawbacks with these schemes, which limited their usefulness to be mostly within the research community. Below, we summarize these drawbacks and discuss our solutions.

- In previous approaches, when disk layout is modified to improve performance, the most frequently-accessed data are always shuffled toward the middle of the disk. However, as frequently-accessed data may change over time, this would require re-shuffling the layout every time the disk access pattern changes. We will show later that this re-shuffling is actually unnecessary in the presence of good reference locality and merely incurs additional migration overheads without any benefit. Instead, blocks should be rearranged only when they are accessed together but lack spatial locality, i.e., those that cause long seeks and long rotational delays between accesses. This can significantly reduce the number of replications we need to perform to effectively modify disk layout.
- When blocks are moved from their original locations they may lose useful data sequentiality. Instead, blocks should

be *replicated* so as to improve both random and sequential accesses. Improving performance for one workload at the expense of another is something we would like to avoid. However, replication consumes additional disk capacity and introduces complexities in maintaining consistency between original disk blocks and their replicas. These issues will be addressed in the next section.

- Using only a narrow region in the middle of the disk to rearrange disk layout is not always practical if the rearrangement is to be done online, because interference with foreground tasks has to be considered. Interference can be significant when foreground tasks are accessing disk regions that are far away from the middle region. We will demonstrate ways to minimize this interference by using the free disk space near current disk head position.

- Some previous techniques reserve a fixed amount of storage capacity just for rearranging disk layout. This is intrusive, as users can no longer utilize the full capacity of their disk. By contrast, using only free disk space is completely transparent to users, and instead of being wasted, free disk space can now be utilized to our benefit. To hide from users the fact that some of their disk capacity is being used to hold replicas, we need to invalidate and free replicated blocks quickly when disk capacity becomes limited. As the amount of free disk space can vary, this technique provides a variable quality of service, where the degree of performance improvement we can achieve depends on the amount of free disk space that is available. However, empirical evidence shows that plenty of free disk space can be found in today's computer systems, so this is rarely a problem.

- Most of previous work is either purely theoretical or based on trace analysis. We implemented a real system and evaluated it using both server-type workloads and common day-to-day single-user workloads, demonstrating significant benefits of using FS^2.

Reducing disk head positioning latencies can make a significant impact on disk performance. This is illustrated in Figure 2, where the disk access time of 4 different disk drives is broken down to various components: transfer time, seek time, and rotational delay. This figure shows that transfer time—the only time during which disk is doing useful work—is dwarfed by seek time and rotational delay. In a random disk workload (as shown in this figure), seek time is by far the most dominating component in a disk access. However, the rotational delay will become the dominating component when the disk access locality reaches 70% (Lumb *et al.* [28] studied this in detail). In addition to improving performance, reduction in seek time (distance) also saves energy as disk dissipates a substantial amount of power when seeking (i.e., when accelerating and decelerating disk heads at 30–40g).

The rest of the paper is organized as follows. Section 2 gives an overview of our system design. Section 3 discusses implementation details of our prototype based on the Ext2 file system. Section 4 presents and analyzes experimental results of FS^2 and compares it with Ext2. Related work is discussed in Section 5. Future research directions are discussed in Section 6, and the paper concludes with Section 7.

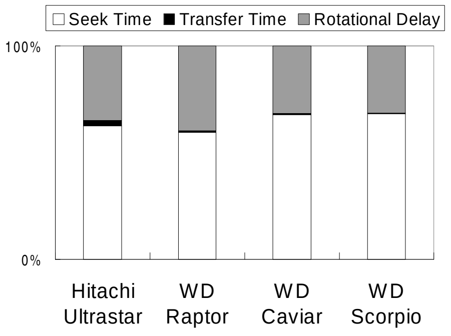

Figure 2: Breakdown of the disk access time in 4 drives, ranging from a top-of-the-line 15000-RPM SCSI drive to a slow 5400-RPM IDE laptop drive.

2. DESIGN

This section details the design of FS^2 on a single-disk system. Most techniques used in building FS^2 on a single-disk system can be directly applied to a disk array (e.g., RAID), yielding a performance improvement similar to that can be achieved on a single disk (in addition to the performance improvements that can be achieved by using RAID). More on this will be discussed in Section 6.

FS^2 differs from other dynamic disk layout techniques due mainly to its exploitation of free disk space, which has a profound impact on our design and implementation decisions. In the next section, we first characterize the availability of free disk space in a large university computing environment and then describe how free disk space can be exploited. Details on how to choose candidate blocks to replicate and how to use free disk space to facilitate the replication of these blocks are given in the following section. Then, we describe the mechanisms we used to keep track of replicas both in memory and on disk so that we can quickly find all alternative locations where data is stored, and then choose the "nearest" one that can be accessed fastest.

2.1 Availability of Free Disk Space

Due to the rapid increase in disk recording density, much larger disks can be built with a fewer number of platters, reducing their manufacturing cost significantly. With bigger and cheaper disks being widely available, we are much less constrained by disk capacity, and hence, are much more likely to leave a significant portion of our disk unused. This is shown from our study of the 242 disks installed on the 22 public servers maintained by the University of Michigan's EECS Department in April 2004. The amount of unused capacity that we have found on these disks is shown in Figure 3. As we have expected, most disks had a substantial amount of unused space—60% of the disks had more than 30% unused capacity, and almost all the disks had at least 10% unused capacity. Furthermore, as file systems may reserve additional disk space for emergency/administrative use, which we did not account for in this study, our observation only gives a lower bound of how much free disk space there really is.

Our observation is consistent with the study done by Douceur *et al.*[9] who reported an average use of only 53% of disk space among 4801 Windows personal computers in a commercial environment. This is a significant waste of resource because, when disk space is not being used, it usually contributes nothing to

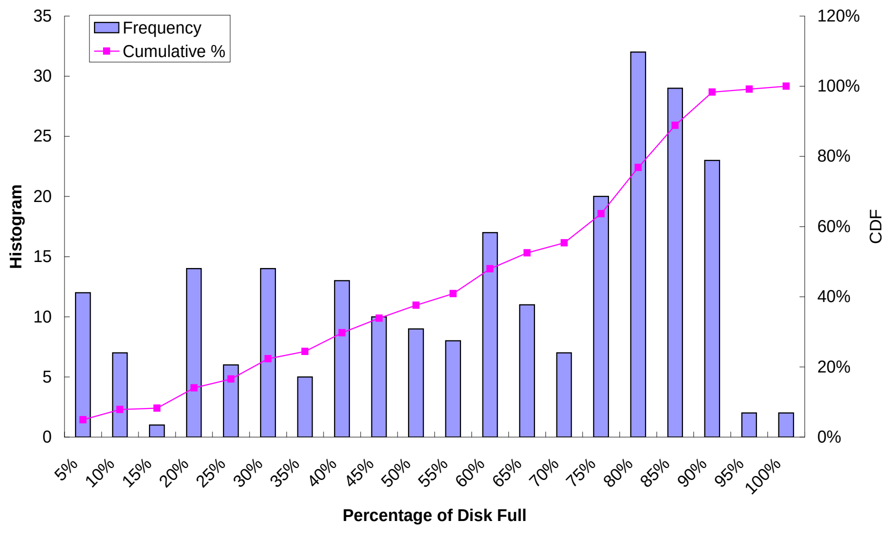

Figure 3: This figure shows the amount of free disk space on 242 disks from 22 public server machines.

users/applications. So, we argue that free disk space can be used to dynamically change disk layout so as to increase disk I/O performance.

Free disk space is commonly maintained by file systems as a large pool of contiguous blocks to facilitate block allocation requests. This makes the use of free disk space attractive for several reasons. First, the availability of these large contiguous regions allows discontiguous disk blocks which are frequently accessed together, to be replicated and aggregated efficiently using large sequential writes. By preserving replicas' adjacency on disk according to the order in which their originals were accessed at runtime, future accesses to these blocks will be made significantly faster. Second, contiguous regions of free disk blocks can usually be found throughout the entire disk, allowing us to place replicas to the location closest to the current disk head's position. This minimizes head positioning latencies when replicating data, and only minimally affects the performance of foreground tasks. Third, using free disk space to hold replicas allows them to be quickly invalidated and then their space to be converted back to free disk space whenever disk capacity becomes tight. Therefore, users can be completely oblivious to how the "free" disk capacity is used, and will only notice performance improvements when more free disk space becomes available.

However, the use of free disk space to hold replicas is not without costs. First, as multiple copies of data may be kept, we will have to keep data and their replicas consistent. Synchronization can severely degrade performance if for each write operation, we have to perform additional writes (possibly multiple) to keep all its replicas consistent. Second, when the user demands more disk space, replicas should be quickly invalidated and their occupied disk space returned to the file system so as to minimize user-perceived delays. However, as this would undo some of the work done previously, we should try to avoid it, if possible. Moreover, as the amount of free disk space can vary over time, deciding how to make use of the available free disk space is also important, especially when there is only a small amount of free space left. These issues will be addressed in the following subsections.

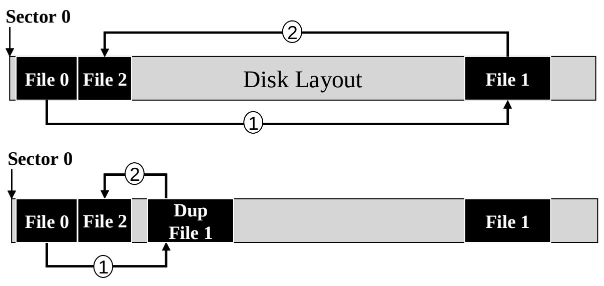

Figure 4: Files are accessed in the order of 0, 1, and 2. Due to the long inter-file distance between File 1 and the other files, long seeks would result if accesses to these files are interleaved. In the bottom figure, we show how the seek distance (time) could be reduced when File 1 is replicated using the free disk space near the other two files.

2.2 Block Replication

We now discuss how to choose which blocks to replicate so as to maximize the improvement of disk I/O performance. We start with an example shown in Figure 4, describing a simple scenario where three files—0, 1, and 2—are accessed, in that order. One can imagine that Files 0 and 2 are binary executables of an application, and File 1 is a shared library, and therefore, may be placed far away from the other two files, as shown in Figure 4. One can easily observe that File 1 is placed poorly if it is to be frequently accessed together with the other two files. If the files are accessed sequentially, where each file is accessed completely before the next, the overhead of seeking is minimal as it would only result in two long seeks—one from 0 to 1 and another from 1 to 2. However, if the accesses to the files are interleaved, performance will degrade severely as many more long-distant seeks would result. To improve performance, one can try to reduce the seek distance between the files by replicating File 1 closer to the other two as shown in the bottom part of Figure 4. Replicating File 1 also reduces rotational delay when its meta-data and data are placed onto the disk consecutively according to the order in which they were accessed. This is illustrated in Figure 5. Reducing rotational delay in some workloads (i.e., those already with a high degree of disk access locality) can be more effective than reducing seek time in improving disk performance.

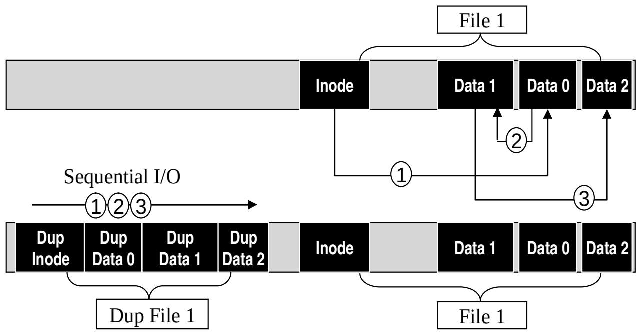

Figure 5: This figure shows a more detailed disk layout of File 1 shown in Figure 4. As file systems tend to place data and their meta-data (and also related data blocks) close to one another, intra-file seek distance (time) is usually short. However, as these disk blocks do not necessarily have to be consecutive, rotational delay will become a more important factor. In the lower portion of the figure, we show how rotational delay can be reduced by replicating File 1 and placing its meta-data and data in the order that they were accessed at runtime.

In real systems, the disk access pattern is more complicated than that shown in the above examples. The examples are mainly to provide an intuition for how seek time and rotational delay can be reduced via replication. We present more practical heuristics used in our implementation in Section 3.

2.3 Keeping Track of Replicas

To benefit from replication of disk blocks, we must be able to find them quickly and decide whether it is more beneficial to access these replicas or their original. This is accomplished by keeping a hash table in memory as shown in Figure 6. It occupies only a small amount of memory: assuming 4KB file system data blocks, a 4MB hash table suffices to keep track of 1GB of replicas on disk. As most of today's computer systems have hundreds of megabytes of memory or more, it is usually not a problem to keep all the hash entries in memory. For each hash entry, we maintain 4 pieces of information: the location of the original block, the location of its replica, the time when the replica was last accessed, and the number of times that the replica was accessed. When free disk space drops below a low watermark (at 10%), the last two items are used to find and invalidate replicas that have not been frequently accessed and were not recently used. At this point, we suspend the replication process. It is resumed only when a high watermark (at 15%) is reached. We also define a critical watermark (at 5%), which triggers a large number of replicas to be invalidated in a batched fashion so a contiguous range of disk blocks can be quickly released to users. This is done by grouping hash entries of replicas that are close to one another on disk so that they are also close to one another in memory, as shown in Figure 6. When the number of free disk blocks drops below the critical watermark, we simply invalidate all the replicas of an entire disk region, one region at a time, until the number of free disk blocks is above the low watermark or when there are no more replicas. During the process, we could have potentially invalidated more valuable replicas and kept less valuable ones. This is acceptable since our primary concern under such a condition is to provide users with the needed disk capacity as quickly as possible.

Given that we can find replicas quickly using the hash table, deciding whether to use an original disk block or one of its replicas is as simple as finding the one closest to the current disk head location. The location of the disk head can be predicted by keeping track of the last disk block that was accessed in the block device driver.

As a result of replication, there may be multiple copies of a data block placed at different locations on disk. Therefore, if the data block is modified, we need to ensure that all of its replicas remain consistent. This can be accomplished by either updating or invalidating the replicas. In the former (updating) option, modifying multiple disk blocks when a single block is modified can be expensive. Not only it generates more disk traffic, but it also incurs synchronization overheads. Therefore, we chose the latter option because both block invalidation and block replication can be done cheaply and in the background.

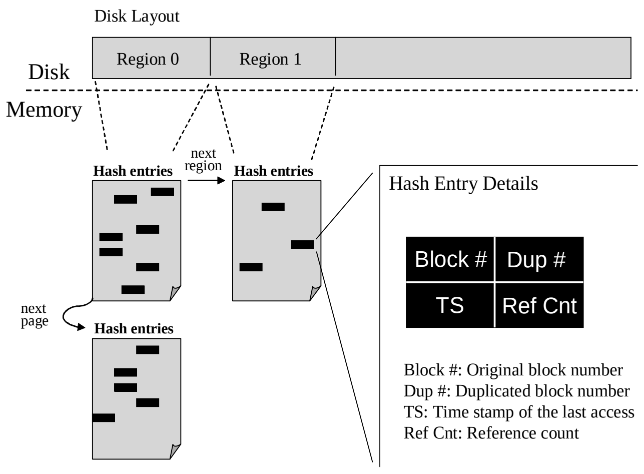

Figure 6: A detailed view of the hash data structure used to keep track of replicas. Hashing is used to speed up the lookup of replicas. Replicas that are close to one another on disk also have their hash entries close to one another in memory, so a contiguous range of replicas can be quickly invalidated and released to users when needed.

3. PROTOTYPE

Our implementation of FS^2 is based on the Ext2 file system [4]. In our implementation, the functionality of Ext2 is minimally altered. It is modified mainly to keep track of replicas and to maintain data consistency. The low-level tasks of monitoring disk accesses, scheduling I/Os between original and replicated blocks, and replicating data blocks are delegated to the lower-level block device driver. Implementation details of that in the block device driver and the file system are discussed in Section 3.1 and Section 3.2, respectively. Section 3.3 describes some of the user-level tools we developed to help users manage the FS^2 file system.

3.1 Block Device Implementation

Before disk requests are sent to the block device driver to be serviced, they are first placed onto an I/O scheduler queue, where they are queued, merged, and rearranged so the disk may be better utilized. Currently, the Linux 2.6 kernel supports multiple I/O schedulers that can be selected at boot time. It is still highly debatable which I/O scheduler performs best, but for workloads that are highly parallel in disk I/Os, the anticipatory scheduler [23] is clearly the winner. Therefore, it is used in both the base Ext2 file system and our FS^2 file system to make a fair comparison.

In our implementation, the anticipatory I/O scheduler is slightly modified so a replica can be accessed in place of its original if the replica can be accessed faster. A disk request is represented by a starting sector number, the size of the request, and the type of the request (read or write). For each read request (write requests are treated differently) that the I/O scheduler passes to the block device driver, we examine each of the disk blocks within the request. There are several different cases we must consider. In the simplest case, where none of the blocks have replicas, we have no choice but to access the original blocks. In the case where all disk blocks within a request have replicas, we access the replicas instead of their originals when (i) the replicas are physically contiguous on disk, and (ii) they are closer to the current disk head's location than their originals. If both criteria are met, to access the replicas instead of their originals, we would need to modify only the starting block number of that disk request. If only a subset of the blocks has replicas, they should not be used as it would break up a single disk request into multiple ones, and in most cases, would take more time. For write accesses, replicas of modified blocks are simply invalidated, as it was explained in Section 2.3.

Aside from scheduling I/Os between original and replicated blocks, the block device driver is also responsible for monitoring disk accesses so it can decide which blocks we should replicate and where the replicas should be placed on disk. We first show how to decide which blocks should be replicated. As mentioned earlier, good candidates are not necessarily frequently-accessed disk blocks, but rather, temporally-related blocks that lack good spatial locality. However, it is unnecessary to replicate all the candidate blocks. This was illustrated previously by the example shown in Figure 4. Instead of replicating all three files, it is sufficient to replicate only File 1—we would like to do a minimal amount of work to reduce mechanical movements. To do so, we first find a *hot region* on disk. A hot region is defined as a small area on disk where the disk head has been most active, and it can change from time to time as workload changes. To minimize disk head positioning latencies, we would like to keep the disk head from moving outside of the hot region. This is accomplished by replicating outside disk blocks that are frequently accessed together with blocks within the hot region and placing them there, if possible. This allows similar disk access patterns to be serviced much faster in future; we achieve shorter seek times due to shorter seek distances and smaller rotational delays because replicas are laid out on disk in the same order that they were previously accessed. In our implementation, we divide a disk into multiple regions,[1] and for each we count the number of times the blocks within the region have been accessed. From the reference count of each disk region, we can approximately locate where the disk head has been most active. This is only an approximation because (i) some disk blocks can be directly accessed from/to the on-disk cache, thus not involving any mechanical movements, and (ii) some read accesses can trigger 2 seek operations if it causes a cache entry conflict and that the cache entry is dirty. However, in the disk traces collected from running various workloads (described in Section 4), we found this heuristic to be sufficient for our purpose.

Although the block device driver can easily locate the hot region on disk and find candidate blocks to replicate, it cannot perform replication by itself because it cannot determine how much free disk space there is, nor can it decide which blocks are free and which are in use. Implicit detection of data liveness from the block device level was shown by Sivathanu *et al.* [13] to be impossible in the Ext2 file system. Therefore, replication requests are forwarded from the block device driver to the file system, where free disk space to hold replicas can be easily located. Contiguous free disk space is used if possible, so that the replicated blocks can be written sequentially to disk. Furthermore, since disk blocks can be replicated more efficiently to regions with more free space, we place more weight on these regions when deciding where to place replicas.

3.2 File System Implementation

Ext2 is a direct descendant of BSD UNIX FFS [12], and is commonly chosen as the default file system by many Linux distributions. Ext2 splits a disk into one or more block groups, each of which contains its own set of inodes (meta-data) and data blocks to enhance data locality and fault-tolerance. More recently, journaling capability was added to Ext2, to produce the Ext3 file system [43], which is backward-compatible to Ext2. We could have easily based our implementation on other file systems, but we chose Ext2 because it is the most popular file system used in today's Linux systems. As with Ext3, FS^2 is backward-compatible with Ext2.

First, we modified Ext2 so it can assist the block device driver to find free disk space near the current disk head's location to place replicas. As free disk space of each Ext2 block group is maintained by a bitmap, large regions of free disk space can be easily found. We also modified Ext2 to assist the block device driver in maintaining data consistency. In Section 3.1, we discussed how data consistency is maintained for write accesses in the block device level. However, not all data consistency issues can be observed and addressed in the block device level. For example, when a disk block with one or more replicas is deleted or truncated from a file, all of its replicas should also be invalidated and released. As the block device driver is completely unaware of block deallocations, we modified the file system to explicitly inform the block device driver of such an event so it can invalidate these replicas and remove them from the hash table.

To allow replicas to persist across restarts, we flush the in-memory hash table to the disk when system shuts down. When the system starts up again, the hash table is read back into the memory. We keep the hash table as a regular Ext2 file using a reserved inode #9 so it cannot be accessed by regular users. To minimize the possibility of data inconsistency that may result from a system crash, we flush any modified hash entries to disk periodically. We developed a set of user-level tools (discussed in Section 3.3) to restore the FS^2 file system back to a consistent state when data consistency is compromised after a crash. An alternative solution is to keep the hash table in flash memory. Flash memory provides a persistent storage medium with low-latency so the hash table can be kept consistent even when the system unexpectedly

[1]For convenience, we simply define a region to be a multiple of Ext2 block groups.

crashes. Given that the size of today's flash memory devices usually ranges between a few hundred megabytes to tens of gigabytes, the hash table can be easily stored.

Ext2 is also modified to monitor the amount of free disk space to prevent replication from interfering with user's normal file system operations. We defined a high (15%), a low (10%), and a critical (5%) watermark to monitor the amount unused disk space and to respond with appropriate actions when a watermark is reached. For instance, when the amount of free disk space drops below the low watermark, we will start reclaiming disk space by invalidating replicas that have been used neither recently nor frequently. If the amount of free disk space drops below the critical watermark, we start batching invalidations for entire disk regions as described in Section 2.3. Moreover, replication is suspended when the amount of free disk space drops below the low watermark, and it is resumed only when the high watermark is reached again.

3.3 User-level Tools

We created a set of user-level tools to help system administrators manage the file system. *mkfs2* is used to convert an existing Ext2 file system to an FS^2 file system, and vice versa. Normally, this operation takes less than 10 seconds. Once an FS^2 file system is created and mounted, it will automatically start monitoring disk traffic and replicating data on its own. However, users with higher-level knowledge about workload characteristics can also help make replication decisions by asking *mkfs2* to statically replicate a set of disk blocks to a specified location on disk. It gives users a knob to manually tune how free disk space should be used.

As mentioned earlier, a file system may crash, resulting in data inconsistency. During system startup, the tool chkfs2 restores the file system to a consistent state if it detects that the file system was not cleanly unmounted, similarly to what Ext2's e2fsck tool does. The time to restore an FS^2 file system back to a consistent state is only slightly longer than that of Ext2, and depends on the number of replicas in the file system.

4. EXPERIMENTAL EVALUATION

We now evaluate FS^2 under some realistic workloads. By dynamically modifying disk layout according to the disk access pattern observed at runtime, seek time and rotational delay are shown to be reduced significantly, which translates to a substantial amount of performance improvement and energy savings. Section 4.1 describes our experimental setup and Section 4.2 details our evaluation methodology. Experimental results of each workload are discussed in each of the subsequent subsections.

4.1 Experimental Setup

Our testbed is set up as shown in Figure 7. The IDE device driver on the testing machine is instrumented, so all of its disk activities are sent via *netconsole* to the monitoring machine, where the activities are recorded. We used separate network cards for running workloads and for collecting disk traces to minimize interference. System configuration of the testing machine and a detailed specification of its disk are provided in Table 1.

Several stress tests were performed to ensure that the system performance is not significantly affected by using netconsole to collect disk traces. We found that the performance was degraded by 5% in the worst case. For common workloads, performance is usually minimally affected. For example, the time to compile a Linux 2.6.7 kernel source code was not impacted at all by recording disk activities via netconsole. Therefore, we believe our tracing mechanism had only a negligible effect on the experimental results.

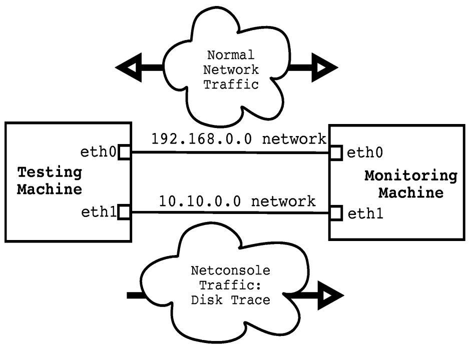

Figure 7: The testbed consists a testing machine for running workloads and a monitoring machine for collecting disk traces. Each machine is equipped with 2 Ethernet cards—one for running workloads and one for collecting disk traces using *netconsole*.

4.2 Evaluation Methodology

For each request in a disk trace, we recorded its start and end times, the starting sector number, the number of sectors requested, and whether it is a read or a write access. The interval between the start and end times is the disk access time, denoted by T_A, which is defined as:

$$T_A = T_s + T_r + T_t,$$

where T_s is the seek time, T_r is the rotational delay, and T_t is the transfer time. Decomposing T_A is necessary for us to understand the implications of FS^2 on each of these components. If the physical geometry of the disk is known (e.g., number of platters, number of cylinders, number of zones, sector per track, etc.), this can be easily accomplished with the information that was logged. Unfortunately, due to increasingly complex hard drive designs and fiercer market competition, disk manufacturers are no longer disclosing such information to the public. They present only a logical

Components	Specification
CPU	Athlon 1.343 GHz
Memory	512MB DDR2700
Network cards	2x3COM 100-Mbps
Disk	Western Digital
Bus interface	IDE
Capacity	78.15 GB
Rotational speed	7200 RPM
Average seek time	9.9 ms
Track-to-track seek time	2.0 ms
Full-stroke seek time	21 ms
Average rotational delay	4.16 ms
Average startup power	17.0 W
Average read/write/idle power	8.0 W
Average seek power	14.0 W

Table 1: System specification of the testing machine.

view of the disk, which, unfortunately, bears no resemblance to the actual physical geometry; it provides just enough information to allow BIOS and drivers to function correctly. As a result, it is difficult to decompose a disk access into its different components. Various tools [37, 42, 14, 1] have been implemented to extract the physical disk geometry experimentally, which usually takes a very long time. Using the same empirical approach, we show that a disk can be characterized in seconds for us to accurately decompose T_A into its various components.

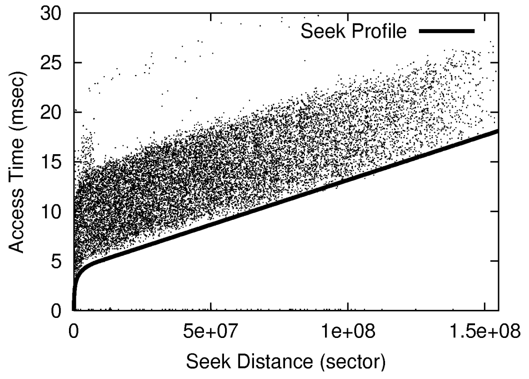

Figure 8: This figure shows logical seek distance versus access time for a Western Digital SE 80GB 7200 RPM drive. Each point in the graph represents a recorded disk access.

Using the same experimental setup as described previously, a random disk trace is recorded from the testing machine. From this trace, we observed a clear relationship between logical seek distance (i.e., the distance between consecutive disk requests in unit of sectors) and T_A, which is shown in Figure 8. The vertical band in this figure is an artifact of the rotational delay's variability, which, for a 7200 RPM disk, can vary between 0 and 8.33 msec (exactly the height of this band). A closer examination of this figure reveals an irregular bump in the disk access time when seek distance is small. This bump appears to reflect the fact that the seek distance is measured in number of sectors, but the number of sectors per track can vary significantly from the innermost cylinder to the outermost cylinder. Thus, the same logical distance can translate to different physical distances, and this effect is more pronounced when the logical seek distance is small.

As T_A is composed of T_s, T_r, and T_t, by removing the variation caused by T_r, we are left with the sum of T_s and T_t, which is represented by the lower envelope of the band shown in Figure 8. As T_t is negligible compared to T_s for a random workload (shown previously in Figure 2), the lower envelope essentially represents the seek profile curve of this disk. This is similar to the seek profile curve, in which, the seek distance is measured in the number of cylinders (physical distance) as opposed to the number of sectors (logical distance). As shown in Figure 8, the seek time is usually linearly related to the seek distance, but for very short distances, it can be approximated as a third order polynomial equation. We observed that for a seek distance of x sectors, T_s (in unit of msec) can be expressed as:

$$T_s = \begin{cases} 8\text{E-}21x^3 - 2\text{E-}13x^2 + 1\text{E-}6x + 1.35 & x < 1.1 \times 10^7 \\ 9\text{E-}8x + 4.16 & \text{otherwise.} \end{cases}$$

As seek distance can be calculated by taking the difference between the starting sector numbers of consecutive disk requests, seek time, T_s, can be easily computed from the above equation. With the knowledge of T_s, we only have to find either T_r or T_t to completely decompose T_A. We found it easier to derive T_t than T_r. To derive T_t, we first measured the effective I/O bandwidth for each of the bit recording zones on the disk using large sequential I/Os (i.e., so we can eliminate seek and rotational delays). For example, on the outer-most zone, the disk's effective bandwidth was measured to be 54 MB/sec. With each sector being 512 bytes long, it would take 9.0 μsec to transfer a single sector from this zone. From a disk trace, we know exactly how many sectors are accessed in each disk request, and therefore, the transfer time of each can be easily calculated by multiplying the number of sectors in the request by the per-sector transfer time. Once we have both T_s and T_t, we can trivially derive T_r. Due to disk caching, some disk accesses will have an access time below the seek profile curve, in which case, since there is no mechanical movement, we treat all of their access time as transfer time.

The energy consumed to service a disk request, denoted as E_A, can be calculated from the values of T_s, T_r and T_t as:

$$E_A = T_s \times P_s + (T_r + T_t) \times P_i,$$

where P_s is the average seek power and P_i is the average idle/read/write power (shown in Table 1). The total energy consumed by a disk can be calculated by adding the sums of all E_A's to the idle energy consumed by the disk.

4.3 The TPC-W Benchmark

The TPC-W benchmark [5], specified by the Transaction Processing Council (TPC), is a transactional web benchmark that simulates an E-commerce environment. Customers browse and purchase products from a web site composed of multiple servers including an application server, a web server and a database server. In our experiment, we ran a MySQL database server and an Apache web server on a single machine. On a separate client machine, we simulated 25 simultaneous customers for 20 minutes. Complying with the TPC-W specification, we assume a mean session time of 15 minutes and a mean think time of 7 seconds for each client. Next, we compare FS^2 with Ext2 with respect to performance and energy consumption.

4.3.1 Ext2: Base System

Figure 9(a1) shows a disk trace collected from running the TPC-W benchmark on an Ext2 file system. Almost all disk accesses are concentrated in several distinct regions of the disk, containing a mixture of the database's data and index files, and the web server's image files. The database files can be large, so a single file can span multiple block groups. Because Ext2 uses a quadratic hash function to choose the next block group from which to allocate disk blocks when the current block group is full, a single database file is often split into multiple segments with each being stored at a different location on the disk. When the file is later accessed, the disk head may have to travel frequently between multiple regions. This is illustrated in Figure 9(a2), showing a substantial number of disk accesses with long seek times.

We observed that the average response time perceived by clients is 750 msec when the benchmark is ran on an Ext2 file system. The average disk access time is found to be 8.2 msec, in which 3.8

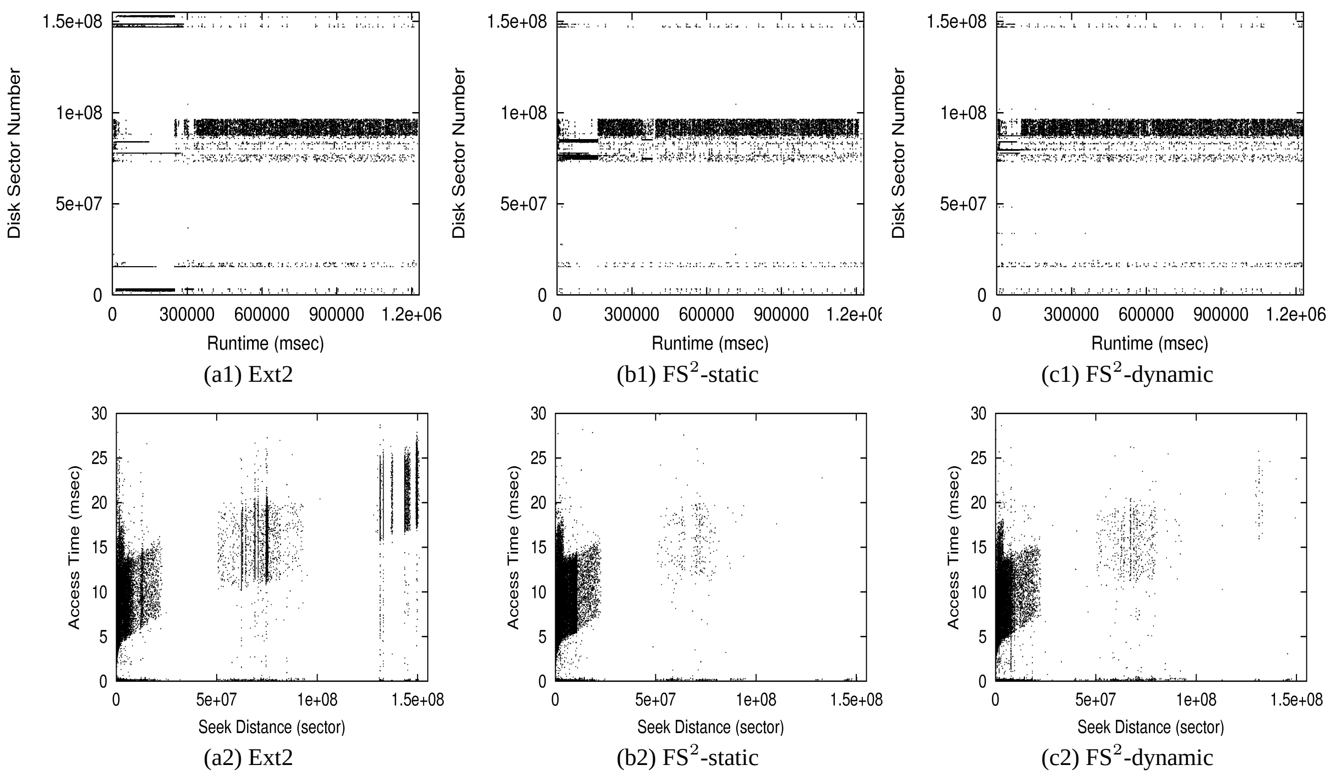

Figure 9: Plots (a1–c1) show the disk sectors that were accessed for the duration of a TPC-W run. Plots (a2–c2) show the measured access times for each request with respect to the seek distance for the TPC-W benchmark. Plots a, b, and c correspond to the results collected on Ext2, FS^2-static, and FS^2-dynamic file systems.

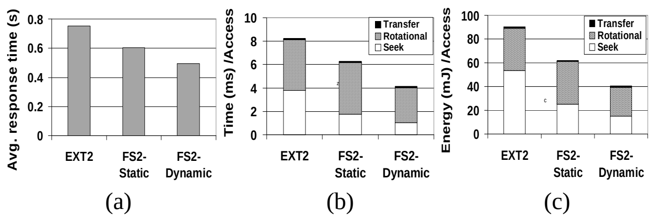

Figure 10: For the TPC-W benchmark, part (a) shows the average response time of each transaction perceived by clients. Part (b) shows the average disk access time and its breakdown. Part (c) shows the average energy consumed by each disk access and its breakdown.

	Disk Busy	Performance Improvement			Energy Improvement
		T_A	T_s	T_r	
FS^2-static	23%	24%	53%	-1.6%	31%
FS^2-dynamic	17%	50%	72%	31%	55%

Table 2: For the TPC-W benchmark, this table shows percentage improvement in performance and energy-consumption for each disk access, with respect to the Ext2 file system, where the disk was busy 31% of the time.

msec is due to seek and 4.3 msec is due to rotation. The average energy consumed by each disk access is calculated to be 90 mJ, in which 54 mJ is due to seek and 36 mJ is due to rotation. These results are plotted in Figure 10 and then summarized in Table 2.

4.3.2 FS^2-Static

From the TPC-W disk trace shown previously, an obvious method to modify the disk layout would be to group all the database and web server files closer to one another on the disk so as to minimize seek distance. To do so, we first find all the disk blocks that belong to either the database or the web server, and then *statically* replicate these blocks (using the *mkfs2* tool) toward the middle of the disk.[2] After the disk layout is modified, we ran the TPC-W workload again, and the resulting disk trace and the access time distribution are plotted in Figures 9(b1) and (b2), respectively. One can see from these figures that as replicas can now be accessed in addition to their originals, most disk accesses are concentrated within a single region. This significantly reduces the range of motions of the disk head for most disk accesses.

As shown in Figure 10, by statically replicating, the average response time perceived by TPC-W clients is improved by 20% over Ext2. The average disk access time is improved by 24%, which is completely due to having shorter seek time (53%). Rotational delay remains mostly unchanged (-1.6%) for reasons to be given shortly. Due to having shorter average seek distance, the energy consumed per disk access is reduced by 31%.

[2]We could have chosen any region, but the middle region of the disk is chosen simply because most of the relevant disk blocks were already placed there. Choosing this region, therefore, would result in the fewest number of replications.

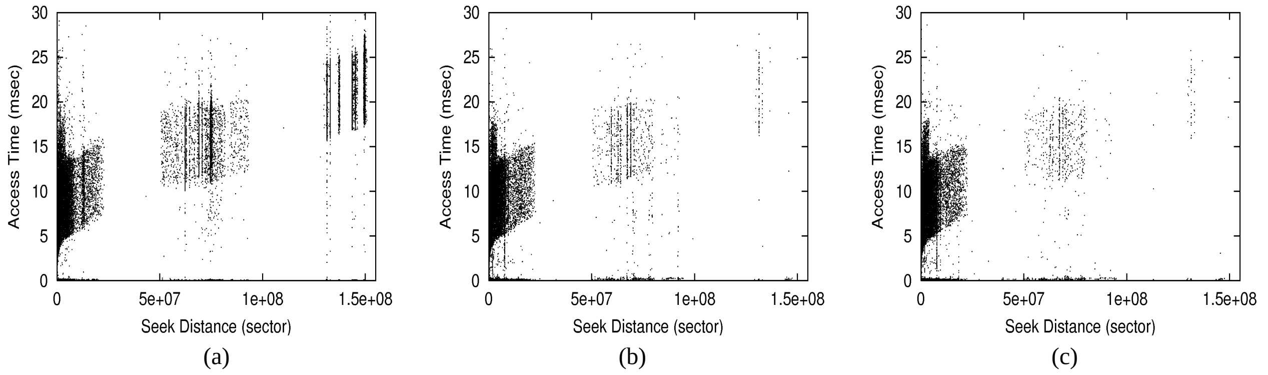

Figure 11: For the TPC-W benchmark, parts (a–c) show disk access time for the 1st, the 2nd, and the 7th FS^2-dynamic run.

4.3.3 *FS^2-Dynamic*

We call the technique described in the previous section *FS^2-static* because it statically modifies the disk layout. Using FS^2-static, seek time was significantly reduced. Unfortunately, as FS^2-static is not effective in reducing rotational delay, disk access time is now dominated more by rotational delay (as shown in Figure 10(b)). This ineffectiveness in reducing rotational delay is attributed to its static aggregation of *all* the disk blocks of database and web server files. It makes the relevant disk blocks much closer to each other on disk, but as long as they are not placed on the same cylinder or track, rotational delay will remain unchanged.

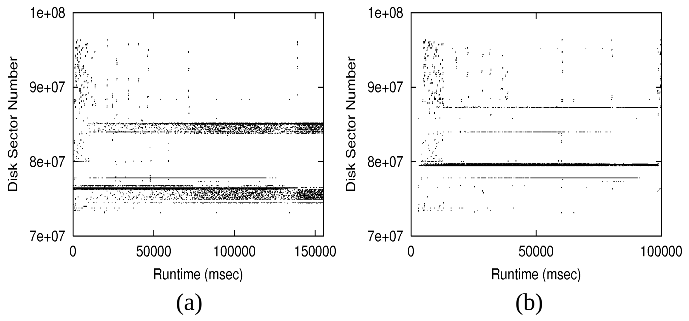

Figure 12: Plots (a) and (b) show the zoom-in section of Figures 9 (b1) and (c1), respectively.

As discussed in Section 2.2, replicating data and placing them *adjacent* to one another according to the order in which they were accessed, not only reduces seek time but also can reduce rotational delay. This is called *FS^2-dynamic* as we dynamically monitor disk accesses and replicate candidate blocks on-the-fly. It is implemented as described in Sections 2 and 3. Starting with no replication, multiple runs were needed for the system to determine which blocks to replicate before observing any significant benefit. The replication overhead was observed to be minimal. Even in the worst case (the first run), the average response time perceived by the TPC-W clients was degraded by only 1.7%. In other runs, the replication overhead is almost negligible. The 7th run of FS^2-dynamic is shown in Figures 9(c1) and (c2). To illustrate the progression between consecutive runs, we show disk access time distributions of the 1st, the 2nd, and the 7th run in Figure 11. During each run, because the middle region of the disk is accessed most, those disk blocks that were accessed in other regions will get replicated here. Consequently, the resulting disk trace looks very similar to that of FS^2-static (shown in Figures 9(b1) and (b2)). There are, however, subtle differences between the two, which enable FS^2-dynamic to significantly outperform its static counterpart.

Figures 12(a) and (b) provide a zoom-in view of the first 150 seconds of the FS^2-static disk trace and the first 100 seconds of the FS^2-dynamic disk trace, respectively. In these two cases, the same amount of data was accessed from the disk, giving a clear indication that the dynamic technique is more efficient in replicating data and placing them onto the disk than the static counterpart. As mentioned earlier, the static technique indiscriminately replicates all statically-related disk blocks together without any regard to how the blocks are temporally related. As a result, the distance between temporally-related blocks becomes smaller but often not small enough for them to be located on the same cylinder or track. On the other hand, as the dynamic technique places replicas onto the disk in the same order that the blocks were previously accessed, it substantially increases the chance for temporally-related data blocks to be on the same cylinder, or even on the same track. From Figure 12, we can clearly see that under FS^2-dynamic, the range of the accessed disk sectors is much narrower than that of FS^2-static.

As a result, FS^2-dynamic makes a 34% improvement in the average response time perceived by the TPC-W clients compared to Ext2. The average disk access time is improved by 50%, which is due to a 72% improvement in seek time and a 31% improvement in rotational delay. FS^2-dynamic reduces both seek time and rotational delay more significantly than FS^2-static. Furthermore, it reduces the average energy consumed by each disk access by 55%. However, because the TPC-W benchmark always runs for a fixed amount of time (i.e., 20 minutes), faster disk access time does not reduce runtime, and hence, the amount of savings in total energy is limited to only 6.7% as idle energy still dominates. In other workloads, when measured over a constant number of disk accesses, as opposed to constant time, the energy savings is more significant.

We introduced FS^2-static and compared it with FS^2-dynamic mainly to illustrate the importance of using online monitoring in reducing rotational delay. As the dynamic technique was shown to perform much better than the static one, we will henceforth consider only FS^2-dynamic and refer to it simply as FS^2 unless specified otherwise. Moreover, only the 7^{th} run of each workload is used when we present results for FS^2 as we want to give enough time for our system to learn frequently-encountered disk access patterns and to modify the disk layout.

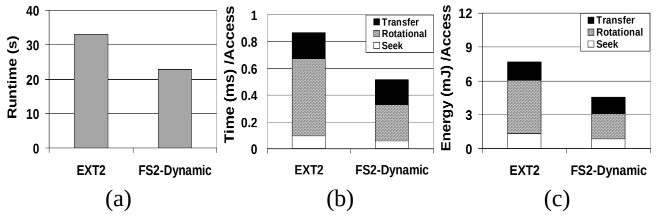

Figure 13: Results for the CVS update command: (a) total runtime, (b) average access time, and (c) energy consumed per access.

	Disk Busy	Performance Improvement			Energy Improvement
		T_A	T_s	T_r	
FS^2	73%	41%	38%	53%	40%

Table 3: A summary of results for the CVS update command. The disk was busy 82% of the time on Ext2.

4.4 CVS

CVS is a versioning control system commonly used in software development environments. It is especially useful for multiple developers who work on the same project and share the same code base. CVS allows each developer to work with his own local source tree and later merge the finished work with a central repository.

CVS update command is one of the most frequently-used commands when working with a CVS-managed source tree. In this benchmark, we ran the command *cvs -q update* in a local CVS directory containing the Linux 2.6.7 kernel source tree. The results are shown in Figure 13. On an Ext2 file system, it took 33 seconds to complete. On FS^2, it took only 23 seconds to complete, a 31% shorter runtime.

The average disk access time is improved by 41% as shown in Figure 13(b). Rotational delay is improved by 53%. Seek time is also improved (37%), but its effect is insignificant as the disk access time in this workload is mostly dominated by rotational delay.

In addition to the performance improvement, FS^2 also saves energy in performing disk I/Os. Figure 13(c) plots the average energy-consumption per disk access for this workload. As shown in Table 3, FS^2 reduces the energy-consumption per access by 46%. The total energy dissipated on Ext2 and FS^2 during the entire execution of the CVS update command is 289 J and 199 J, respectively, making a 31% energy savings.

4.5 Starting X Server and KDE

The *startx* command starts the X server and the KDE windows manager on our testing machine. As shown in Figure 14(a), the startx command finished 4.6 seconds faster on FS^2 than Ext2, yielding a 16% runtime reduction. Even though the average disk access time in this workload is significantly improved (44%), which comes as a result of a 53% shorter average seek time and 47% shorter rotational delay (shown in Table 4), the runtime is

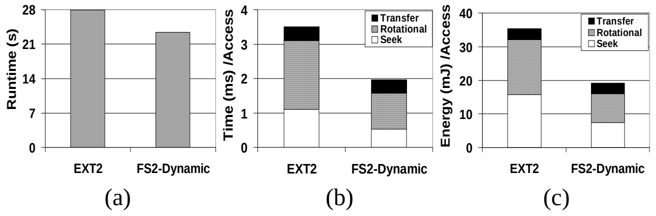

Figure 14: Results for starting X server and KDE windows manager: (a) total runtime, (b) average access time, and (c) energy consumed per access.

	Disk Busy	Performance Improvement			Energy Improvement
		T_A	T_s	T_r	
FS^2	26%	44%	53%	47%	46%

Table 4: A summary of results for starting X server and KDE windows manager. The disk was busy 37% of the time on Ext2.

only moderately improved and much less than what we had initially expected. The reason is that this workload is not disk I/O-intensive, i.e., much less than the CVS workload. For the CVS workload, the disk was busy for 82% of the time, whereas for this workload, the disk was busy for only 37% of the time.

In addition to improving performance, FS^2 also reduces the energy consumed per disk access for the startx command as shown in Figure 14(c). Starting the X server is calculated to dissipate a total energy of 249 J on Ext2 and 201 J on FS^2, making a 19% savings in energy.

4.6 SURGE

SURGE is a network workload generator designed to simulate web traffic by imitating a collection of users accessing a web server [3]. To best utilize the available computing resources, many web servers are supporting multiple virtual hosts. To simulate this type of environment, we set up an Apache web server supporting two virtual hosts servicing static contents on ports 81 and 82 of the testing machine. Each virtual host supports up to 200 simultaneous clients in our experiment.

FS^2 reduced the average response time of the Apache web server by 18% compared to Ext2 as illustrated in Figure 15(a). This improvement in the average response time was made possible despite the fact that the disk was utilized for only 13% of the time. We tried to increase the number of simultaneous clients to 300 per virtual host to observe what would happen if the web server was busier, but the network bandwidth bottleneck on our 100 Mb/sec LAN did not allow us to drive up the workload. With more simultaneous clients, we expect to see more performance improvements by FS^2. However, we show that even for non-disk-bound workloads, performance can still be improved reasonably well.

On average, a disk access takes 68% less time on FS^2 than on Ext2. In this particular workload, reductions in seek time and rotational delay contributed almost equally in lowering the disk access time, as shown in Table 5.

The total energy consumed was reduced by merely 1.9% on FS^2 because this workload always runs for a fixed amount time like the TPC-W workload. There might be other energy-saving opportunities. In particular, FS^2 allows disk to become idle more often—in the TPC-W workload, the disk was idle 70% of the time

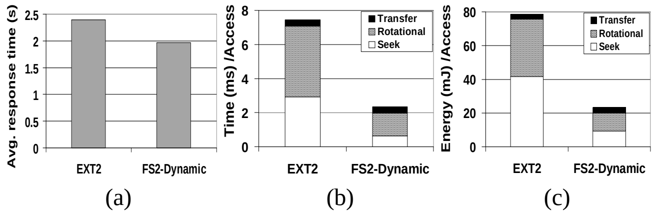

Figure 15: Results for SURGE: (a) average response time, (b) average access time, and (c) energy consumed per access.

	Disk	Performance Improvement			Energy
	Busy	T_A	T_s	T_r	Improvement
FS^2	4.2%	69%	78%	68%	71%

Table 5: A summary of results for running SURGE. The disk was busy 13% of the time on Ext2.

on Ext2 and 83% on FS^2, while in the CVS workload, the disk was idle 18% of the time on Ext2 and 27% on FS^2. Combined with traditional disk power-management techniques, which save energy during idle intervals, more energy-saving opportunities can be exploited.

5. RELATED WORK

File System: FFS [12] and its successors [4, 43] improve disk bandwidth and latency by placing related data objects (e.g., data blocks and inodes) near each other on disk. Albeit effective in initial data placement, they do not consider dynamic access patterns. As a result, poorly-placed disk blocks can significantly limit the degree of performance improvements [2, 36]. Based on FFS, Ganger *et al.* [17] proposed a technique that improves performance for small objects by increasing their adjacency rather than just locality. Unfortunately, they also suffer from the same problem as FFS.

In the Log-structured File System (LFS) [35], disk-write performance can be significantly improved by placing blocks that are written close to one another in time to large contiguous segments on disk. Similarly, FS^2 places (replicas of) blocks that are read close to one another in time to large contiguous free disk space. Tuning LFS by putting active segments on higher-bandwidth outer cylinders of the disk can further improve write performance [45]. However, due to segment cleaning overheads and LFS's inability to improve read performance (and most workloads are dominated by reads), there is no clear indication that it can outperform the file systems in the FFS family.

The closest work related to FS^2 are Hot File Clustering used in the Hierarchical File System (HFS) [16] and the Smart File System [41]. These file systems monitor file access patterns at runtime and move frequently-accessed files to a reserved area on disk. There are several major advantages of FS^2 over these other techniques. First, the use of block granularity allows us to optimize for all files as opposed to just the small ones. It also gives us opportunities to place meta-data and data blocks *adjacent* to one another according to the order that they were previously accessed, which, as we have shown and also as indicated in [17], is very important to the reduction of rotational delay. Second, moving data can potentially break sequentiality. Instead, we replicate disk blocks, so both sequential and random disk I/Os can benefit. Third, using a fixed location on disk to rearrange disk layout will have benefit only under light load [25]. Using free disk space, we show that even under heavy load, significant benefits can be obtained while keeping replication overhead low. Fourth, reserving a fixed amount of disk space to rearrange disk layout is intrusive, which can be prevented by using only the free disk space as we have done in this paper.

Adaptive Disk Layout: Early adaptive disk layout techniques have been mostly studied either theoretically [46] or via simulation [10, 30, 36, 44]. It has been shown for random disk accesses that the *organ pipe* heuristic, which places the most frequently-accessed data to the middle of the disk, is optimal [46]. However, as real workloads do not exhibit random disk patterns, the organ pipe placement is not always optimal, and for various reasons, far from being practical. To adapt the organ pipe heuristic to realistic workloads, Vongsathorn and Carson [44] proposed *cylinder shuffling*. In their approach, the access frequency of each cylinder is maintained, and based on this usage frequency, cylinders are reordered using the organ pipe heuristic at the end of each day. Ruemmler and Wilkes [36] suggested shuffling in units of blocks instead of cylinders, and it was shown to be more effective. In all these techniques, it was assumed that the disk geometry is known and can be easily simulated, but due to increasingly complex hard disk designs, disk geometry is now mostly hidden. There are tools [1, 14, 37, 42] that can be used to extract the physical disk geometry experimentally, but the extraction usually takes a long time and the extracted information can be too tedious to be used practically at runtime. Our work only uses disk logical geometry, and we have shown this to be sufficient in practice by verifying it with the extracted disk physical geometry.

Akyurek and Salem [2] were the first to report block shuffling in a real system. Unfortunately, other than its use of block granularity instead of file granularity, this technique is very similar to HFS and Smart File System, and therefore, suffers from the same pitfalls as HFS and Smart File System.

I/O Scheduling: Disk performance can also be improved by means of better I/O scheduling. Seek time can be reduced by using strategies such as SSTF and SCAN [6], C-SCAN [38], and LOOK [29]. Rotational delay can be reduced by taking the approaches in [21, 34, 39, 24]. An anticipatory I/O scheduler [23] is used to avoid hasty I/O scheduling decisions by introducing an additional wait time. These techniques are orthogonal to our work.

Energy Efficiency: Due mainly to higher rotational speeds and heavier platters, disk drives are consuming more energy. Various techniques [8, 7, 19, 26] have been developed to save energy by exploiting disk idleness. However, due to increasing utilization of disk drives, most idle periods are becoming too small to be exploited for saving energy [20, 47]. Gurumurthi *et al.* [20] proposed a Dynamic RPM (DRPM) disk design, and in their simulation, it was shown to be effective in saving energy even when the disk is fairly busy. However, as DRPM disks are not currently pursued by any disk manufacturers, we cannot benefit from this new architecture. In this paper, we lowered power dissipation by reducing seek distance. Similar to the DRPM design, power can be reduced even when disk is busy. However, unlike DRPM, FS^2 does not suffer from any performance degradation. On the contrary, FS^2 actually improves performance while saving energy.

Utilization of Free Disk Space: Free disk space has been exploited by others in the past for different purposes. The Elephant Filesystem [11] uses free disk space to automatically store old versions of files to avoid explicit user-initiated version control. In their system, a cleaner is used to free older file versions to reclaim disk space, but it is not allowed to free *Landmark* files. In FS^2, all free space used for duplication can be reclaimed as replicas are only used to enhance performance and invalidating them will not compromise the correctness of the system.

6. DISCUSSION

6.1 Degree of Replication

Our implementation of FS^2 allows system administrators to set the maximum degree of replication; in an n-replica system, a disk block can be replicated at most n times. According to this definition, traditional file systems are 0-replica systems. There are certain tradeoffs in choosing the parameter n. Obviously, with a large n, replicas can potentially consume more disk space. Also, there is a cost to invalidate or free replicas. However, allowing a larger degree of replication can more significantly improve performance under certain situations. One simple example is the placement of shared libraries. Placing multiple copies of a shared library close to each application image that make use of it, application startup time can be reduced. It is interesting to find the “optimal” degree of replication for a certain disk access pattern. However, in the workloads that we have studied, due to lack of data/code sharing, the additional benefit from having multiple replicas per disk block would be minimal, and therefore, we simply used a 1-replica system in our evaluation. Lo [18] explored the effect of degree of replication in more details through a trace-driven analysis.

6.2 Reducing Replication Overhead

Replicating data in free disk space that is near the current disk head location minimizes interference with foreground tasks. Using *freeblock scheduling* [27, 28], it is possible to further reduce this overhead by writing replicas to disk only during the rotational delay. However, as indicated by the experimental results, our heuristic is already fairly effective in keeping the replication overhead low.

6.3 Multi-Disk Issues

The most straightforward way to improve disk performance is to spread data across more disks. Redundant Array of Inexpensive Disks (RAID) was first introduced by Patterson *et al.* [33] as a way to improve aggregated I/O performance by exploiting parallelism between multiple disks, and since then it has become widely used.

FS^2 is operating orthogonally to RAID. The techniques that we have used to improve performance on a single disk can be directly applied to each of the disks in a RAID. Therefore, the same amount of performance improvement that we have seen before can still be achieved here. However, the problem gets more interesting when free disk space from multiple disks can be utilized. It can be used not only to improve performance, but also to enhance the fault-tolerance provided by RAID. For example, a traditional RAID-5 system can sustain only one disk failure before data loss. However, if we are able to use free disk space on one disk to hold recently-modified data on other disks, it may be possible that even with more than one disk failure, all data can still be recovered—recently-modified data can be recovered from replicas on non-faulty disks and older data can be recovered from a nightly or weekly backup system.

This introduces another dimension where free disk space can be utilized to our advantage. Now, we can use free space of a disk to hold either replicas of its own disk blocks or replicas from other disks. Both have performance benefits, but holding replicas for other disks can also improve fault-tolerance. These issues are beyond the scope of this paper. We are currently investigating these issues and building a RAID-based prototype.

7. CONCLUSION

In this paper, we presented the design, implementation and evaluation of a new file system, called FS^2, which contains a runtime component responsible for dynamically reorganizing disk layout. The use of free disk space allows a flexible way to improve disk I/O performance and reduce its power dissipation, while being nonintrusive to users.

To evaluate the effectiveness of FS^2, we conducted experiments using a range of workloads. FS^2 is shown to improve the client-perceived response time by 34% in the TPC-W benchmark. In SURGE, the average response time is improved by 18%. Even for everyday tasks, FS^2 is shown to provide significant performance benefits: the time to complete a CVS update command is reduced by 31% and the time to start an X server and a KDE windows manager is reduced by 16%. As a result of reducing seek distance, energy dissipated due to seeking is also reduced. Overall, FS^2 is shown to have reduced the total energy consumption of the disk by 1.9–15%.

We are currently working on an extension of this technique that complements the level of fault-tolerance achieved by using RAID. In such a system, not only is free disk space useful in improving performance and energy savings, it may also be possible to provide additional fault-tolerance in addition to that provided by RAID, which can better safeguard the data stored on a RAID system when multiple failures occur.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Mike Burrows, for their comments that helped us improve this paper. We also thank Karthick Rajamani at IBM Austin Research Lab for helping us setting up the TPC-W workload, Howard Tsai at The University of Michigan for helping us setting up the SURGE workload. This work is partially funded by AFOSR under grants F49620-01-1-0120 and FA9550-05-0287.

9. REFERENCES

[1] M. Aboutabl and A. Agrawala and J. Decotignie, Temporally Determinate Disk Access: An Experimental Approach, *Technical Report, CS-TR-3752*, 1997.

[2] Sedat Akyurek and Kenneth Salem, Adaptive Block Rearrangement, *Computer Systems*, 13(2): 89–121, 1995.

[3] P. Barford and M. Crovella, Generating Representative Web Workloads for Network and Server Performance Evaluation, *Proceedings of the ACM Measurement and Modeling of Computer Systems*, 151–160, 1998.

[4] R. Card and T. Ts’o and S. Tweedle, Design and Implementation of the Second Extended Filesystem, *First Dutch International Symposium on Linux*, 1994.

[5] Transactional Processing Performance Council, http://www.tpc.org/tpcw.

[6] P. J. Denning, Effects of Scheduling on File Memory Operations, *AFIPS Spring Joint Computer Conference*, 9–21, 1967.

[7] F. Douglis and P. Krishnan and B. Bershad, Adaptive Disk Spin-down Policies for Mobile Computers, *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, 1995.

[8] F. Douglis and P. Krishnan and B. Marsh, Thwarting the Power-Hungry Disk, *USENIX Winter*, 292–306, 1994.

[9] John Douceur and William Bolosky, A Large-Scale Study of File-System Contents, *ACM SIGMETRICS Performance Review*, 59–70, 1999.

[10] Robert English and Alexander Stepanov, Loge: A Self-Organizing Disk Controller, *Proceedings of the Winter 1992 USENIX Conference*, 1992.

[11] Douglas Santry *et al.*, Deciding When to Forget in the Elephant File System, *ACM Symposium on Operating Systems Principles*, 110–123, 1999.

[12] M. K. McKusick *et al.*, A Fast File System for UNIX, *ACM Transactions on Computing Systems (TOCS)*, 2(3), 1984.

[13] Muthian Sivathanu *et al.*, Life or Death at Block Level, *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 379–394, 2004.

[14] Zoran Dimitrijevic *et al.*, Diskbench: User-level Disk Feature Extraction Tool, *Technical Report, UCSB*, 2004.

[15] Linux Filesystem Performance Comparison for OLTP, http://oracle.com/technology/tech/linux/pdf/Linux-FS-Performance-Comparison.pdf.

[16] HFS Plus Volume Format, http://developer.apple.com/technotes/tn/tn1150.html.

[17] Greg Ganger and Frans Kaashoek, Embedded Inodes and Explicit Groups: Exploiting Disk Bandwidth for Small Files, *USENIX Annual Technical Conference*, 1–17. 1997.

[18] Sai-Lai Lo, Ivy: a Study on Replicating Data for Performance Improvement, *HP Technical Report, HPL-CSP-90-48*, 1990.

[19] R. A. Golding and P. Bosch and C. Staelin and T. Sullivan and J. Wilkes, Idleness is Not Sloth, *USENIX Winter*, 201–212, 1995.

[20] Gurumurthi *et al.*, DRPM: Dynamic Speed Control for Power Management in Server Class Disks, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003.

[21] L. Huang and T. Chiueh, Implementation of a Rotation Latency Sensitive Disk Scheduler, *Technical Report, ECSL-TR81*, 2000.

[22] IBM DB2, http://www-306.ibm.com/software/data/db2.

[23] Sitaram Iyer and Peter Druschel, Anticipatory Scheduling: A Disk Scheduling Framework To Overcome Deceptive Idleness in Synchronous I/O, *18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[24] D. Jacobson and J. Wilkes, Disk Scheduling Algorithms Based on Rotational Position, *HP Technical Report, HPL-CSP-91-7*, 1991.

[25] Richard King, Disk Arm Movement in Anticipation of Future Requests, *ACM Transaction on Computer Systems*, 214–229, 1990.

[26] K. Li and R. Kumpf and P. Horton and T. E. Anderson, Quantitative Analysis of Disk Drive Power Management in Portable Computers, *USENIX Winter*, 279–291, 1994.

[27] C. Lumb and J. Schindler and G. Ganger, Freeblock Scheduling Outside of Disk Firmware, *Conference on File and Storage Technologies (FAST)*, 275–288, 2002.

[28] C. Lumb and J. Schindler and G. Ganger and D. Nagle and E. Riedel, Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives, *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2000.

[29] A. G. Merten, Some Quantitative Techniques for File Organization, *Ph.D. Thesis*, 1970.

[30] David Musser, Block Shuffling in Loge, *HP Technical Report CSP-91-18*, 1991.

[31] MySQL, http://www.mysql.com.

[32] Douglas Orr and Jay Lepreau and J. Bonn and R. Mecklenburg, Fast and Flexible Shared Libraries, *USENIX Summer*, 237–252, 1993.

[33] D. Patterson and G. Gibson and R. Katz, A Case for Redundant Arrays of Inexpensive Disks (RAID), *Proceedings of ACM SIGMOD*, 109–116, 1988.

[34] Lars Reuther and Martin Pohlack, Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS), *IEEE Real-Time Systems Symposium*, 2003.

[35] M. Rosenblum and J. Ousterhout, The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, 26–52, 1992.

[36] C. Ruemmler and J. Wilkes, Disk Shuffling, *HP Technical Report, HPL-CSP-91-30*, 1991.

[37] J. Schindler and G. Ganger, Automated Disk Drive Characterization, *Technical Report CMU-CS-99-176*, 1999.

[38] P. H. Seaman and R. A. Lind and T. L. Wilson, An Analysis of Auxiliary-Storage Activity, *IBM System Journal*, 5(3): 158–170, 1966.

[39] M. Seltzer and P. Chen and J. Ousterhout, Disk Scheduling Revisited, *USENIX Winter*, 313–324, 1990.

[40] SQL Server, http://www.microsoft.com/sql/default.mspx.

[41] C. Staelin and H. Garcia-Molina, Smart Filesystems, *USENIX Winter*, 45–52, 1991.

[42] N. Talagala and R. Arpaci-Dusseau and D. Patterson, Microbenchmark-based Extraction of Local and Global Disk Characteristics, *Technical Report CSD-99-1063*, 1999.

[43] Stephen Tweedie, Journaling the Linux ext2fs Filesystem, *LinuxExpo*, 1998.

[44] P. Vongsathorn and S. D. Carson, A System for Adaptive Disk Rearrangement, *Software Practice Experience*, 20(3): 225–242, 1990.

[45] J. Wang and Y. Hu, PROFS – Performance-Oriented Data Reorganization for Log-structured File System on Multi-Zone Disks, *The 9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems*, 285–293, 2001.

[46] C. K. Wong, Algorithmic Studies in Mass Storage Systems *Computer Sciences Press*, 1983.

[47] Qingbo Zhu *et al.*, Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management, *The 10th International Symposium on High Performance Computer Architecture (HPCA-10)*, 2004.