TFS: A Transparent File System for Contributory Storage

James Cipar Mark D. Corner Emery D. Berger

Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{*jcipar, mcorner, emery*}*@cs.umass.edu*

Abstract

Contributory applications allow users to donate unused resources on their personal computers to a shared pool. Applications such as SETI@home, Folding@home, and Freenet are now in wide use and provide a variety of services, including data processing and content distribution. However, while several research projects have proposed contributory applications that support peer-to-peer storage systems, their adoption has been comparatively limited. We believe that a key barrier to the adoption of contributory storage systems is that contributing a large quantity of local storage interferes with the principal user of the machine.

To overcome this barrier, we introduce the Transparent File System (TFS). TFS provides background tasks with large amounts of unreliable storage—all of the currently available space—without impacting the performance of ordinary file access operations. We show that TFS allows a peer-to-peer contributory storage system to provide 40% more storage at twice the performance when compared to a user-space storage mechanism. We analyze the impact of TFS on replication in peer-to-peer storage systems and show that TFS does not appreciably increase the resources needed for file replication.

1 Introduction

Contributory applications allow users to donate unused resources from their personal computers to a shared pool. These applications harvest idle resources such as CPU cycles, memory, network bandwidth, and local storage to serve a common distributed system. These applications are distinct from other peer-to-peer systems because the resources being contributed are not directly consumed by the contributor. For instance, in Freenet [8], all users contribute storage, and any user may make use of the storage, but there is no relationship between user data and contributed storage. Contributory applications in wide use include computing efforts like Folding@home [17] and anonymous publishing and content distribution such as Freenet [8]. The research community has also developed a number of contributory applications, including distributed backup and archival storage [30], server-less network file systems [1], and distributed web caching [11]. However, the adoption of storage-based contributory applications has been limited compared to those that are CPU-based.

Two major barriers impede broader participation in contributory storage systems. First, existing contributory storage systems degrade normal application performance. While *transparency*—the effect that system performance is as if no contributory application is running—has been the goal of other OS mechanisms for network bandwidth [34], main memory [7], and disk scheduling [19], previous work on contributory storage systems has ignored its local performance impact. In particular, as more storage is allocated, the performance of the user's file system operations quickly degrades [20].

Second, despite the fact that end-user hard drives are often half empty [10, 16], users are generally reluctant to relinquish their free space. Though disk capacity has been steadily increasing for many years, users view storage space as a limited resource. For example, three of the Freenet FAQs express the implicit desire to donate less disk space [12]. Even when users are given the choice to limit the amount of storage contribution, this option requires the user to decide *a priori* what is a reasonable contribution. Users may also try to donate as little as possible while still taking advantage of the services provided by the contributory application, thus limiting its overall effectiveness.

Contributions: This paper presents the Transparent File System (TFS), a file system that can contribute 100% of the idle space on a disk while imposing a negligible performance penalty on the local user. TFS operates by storing files in the free space of the file system so that they are invisible to ordinary files. In essence,

normal file allocation proceeds as if the system were not contributing any space at all. We show in Section 5 that TFS imposes nearly no overhead on the local user. TFS achieves this both by minimizing interference with the file system's block allocation policy and by sacrificing persistence for contributed space: normal files may overwrite contributed space at any time. TFS takes several steps that limit this unreliability, but because contributory applications are already designed to work with unreliable machines, they behave appropriately in the face of unreliable files. Furthermore, we show that TFS does not appreciably impact the bandwidth needed for replication. Users typically create little data in the course of a day [4], thus the erasure of contributed storage is negligible when compared to the rate of machine failures.

TFS is especially useful for replicated storage systems executing across relatively stable machines with plentiful bandwidth, as in a university or corporate network. This environment is the same one targeted by distributed storage systems such as FARSITE [1]. As others have shown previously, for high-failure modes, such as wide-area Internet-based systems, the key limitation is the bandwidth between nodes, not the total storage. The bandwidth needed to replicate data after failures essentially limits the amount of storage the network can use [3]. In a stable network, TFS offers substantially more storage than dynamic, user-space techniques for contributing storage.

Organization: In Section 2, we first provide a detailed explanation of the interference caused by contributory applications, and discuss current alternatives for contributing storage. Second, we present the design of TFS in Section 3, focusing on providing transparency to normal file access. We describe a fully operating implementation of TFS. We then explain in Section 4 the interaction between machine reliability, contributed space, and the amount of storage used by a contributory storage system. Finally, we demonstrate in Section 5 that the performance of our TFS prototype is on par with the file system it was derived from, and up to twice as fast as user-space techniques for contributing storage.

2 Interference from Contributing Storage

All contributory applications we are aware of are configured to contribute a small, fixed amount of storage—the contribution is small so as not to interfere with normal machine use. This low level of contribution has little impact on file system performance and files will generally only be deleted by the contributory system, not because the user needs storage space. However, such small, fixed-size contributions limit contribution to small-scale storage systems.

Instead of using static limits, one could use a dynamic system that monitors the amount of storage used by local applications. The contributory storage system could then use a significantly greater portion of the disk, while yielding space to the local user as needed. Possible approaches include the watermarking schemes found in Elastic Quotas [18] and FS^2 [16]. A contributory storage system could use these approaches as follows: whenever the current allocation exceeds the maximum watermark set by the dynamic contribution system, it could delete contributory files until the contribution level falls below a lower watermark.

However, if the watermarks are set to comprise all free space on the disk, the file system is forced to delete files synchronously from contributed storage when writing new files to disk. In this case, the performance of the disk would be severely degraded, similar to the synchronous cleaning problem in LFS [31]. For this reason, Elastic Quotas and FS^2 use more conservative watermarks (e.g., at most 85%), allowing the system to delete files lazily as needed.

Choosing a proper watermark leaves the system designer with a trade-off between the amount of storage contributed and local performance. At one end of the spectrum, the system can contribute little space, limiting its usefulness. At the other end of the spectrum, local performance suffers.

To see why local performance suffers, consider the following: as a disk fills, the file system's block allocation algorithm becomes unable to make ideal allocation decisions, causing fragmentation of the free space and allocated files. This fragmentation increases the seek time when reading and writing files, and has a noticeable effect on the performance of disk-bound processes. In an FFS file system, throughput can drop by as much as 77% in a file system that is only 75% full versus an empty file system [32]—the more storage one contributes, the worse the problem becomes. The only way to avoid this is to maintain enough free space on the disk to allow the allocation algorithm to work properly, but this limits contribution to only a small portion of the disk.

Though some file systems provide utilities to defragment their disk layout, these utilities are ineffective when there is insufficient free space on the file system. For instance, the defragmentation utility provided with older versions of Microsoft Windows will not even attempt to defragment a disk if more than 85% is in use. On modern Windows systems, the defragmentation utility will run when the disk is more than 85% full, but will give a warning that there is not enough free space to defragment properly [22]. When one wants to contribute all of the free space on the disk, they will be unable to meet these requirements of the defragmentation utility.

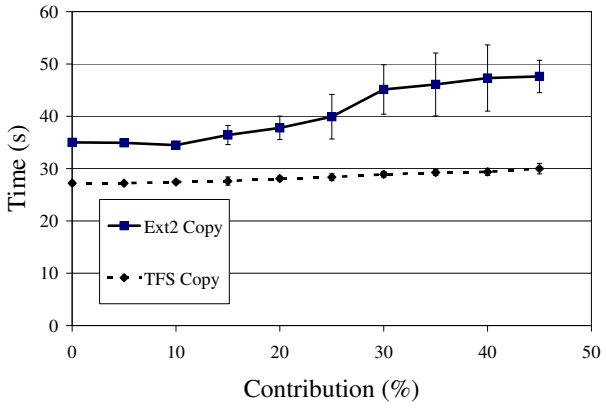

Figure 1: The time required to perform a series of file system operations while contributing different amounts of storage. As the amount of contributed space increases, the time it takes for Ext2 to complete the experiment also increases. However, the performance of TFS stays nearly constant. Error bars represent standard deviation.

Any user-space scheme to manage contributed storage will be plagued by the performance problems introduced by the contributed storage—filling the disk with data inevitably slows access. Given a standard file system interface, it is simply not possible to order the allocation of data on disk to preserve performance for normal files. As Section 3 shows, by incorporating the mechanisms for contribution into the file system itself, TFS maintains file system performance even at high levels of contribution.

Figure 1 depicts this effect. This figure shows the time taken to run the copy phase of the Andrew Benchmark on file systems with different amounts of space being consumed. The full details of the benchmark are presented in Section 5. As the amount of consumed space increases, the time it takes to complete the benchmark increases. We assume that the user is using 50% of the disk space for non-contributory applications, which corresponds to results from a survey of desktop file system contents [10]. The figure shows that contributing more than 20% of the disk space will noticably affect the file system's performance, even if the contributed storage would have been completely idle. As a preview of TFS performance, note that when contributing 35% of the disk, TFS is *twice as fast* as Ext2 for copying files.

3 Design and Implementation of TFS

The Transparent File System (TFS) allows users to donate storage to distributed storage systems with minimal performance impact. Because the block allocation policy is one of the primary determinants of file system performance, designers have devoted considerable attention to tuning it. Accordingly, deviating from that policy can result in a loss of performance. The presence of data on the file system can be viewed as an obstruction which causes a deviation from the default allocation policy. The goal of TFS is to ensure the *transparency* of contributed data: the presence of contributed storage should have no measurable effect on the file system, either in performance, or in capacity. We use the term *transparent files* for files which have this property, and *transparent data* or *transparent blocks* for the data belonging to such files. A *transparent application* is an application which strives to minimize its impact on other applications running on the same machine, possibly at the expense of its own performance.

Section 3.1 shows how we achieve transparency with respect to block allocation in the context of a popular file system, Ext2 [5]. Ext2 organizes data on disk using several rules of thumb that group data on disk according to logical relationships. As we show in Section 5, TFS minimally perturbs the allocation policy for ordinary files, yielding a near-constant ordinary file performance regardless of the amount of contributed storage.

In exchange for this performance, TFS sacrifices file persistence. When TFS allocates a block for an ordinary file, it treats free blocks and transparent blocks the same, and thus may overwrite transparent data. Files marked transparent may be overwritten and deleted at any time. This approach may seem draconian, but because replicated systems already deal with the failure of hosts in the network, they can easily deal with the loss of individual files. For instance, if one deletes a file from a BitTorrent peer, other peers automatically search for hosts that have the file.

The design of TFS is centered around tracking which blocks are allocated to which kind of file, preserving persistence for normal user files, and detecting the overwriting of files in the contributed space. As the file system is now allowed to overwrite certain other files, it is imperative that it not provide corrupt data to the contribution system, or worse yet, to the user. While our design can preserve transparency, we have also made several small performance concessions which have minimal effect on normal file use, but yield a better performing contribution system. Additionally, file systems inevitably have hot spots, possibly leading to continuous allocation and deallocation of space to and from contribution. These hot spots could lead to increased replication traffic elsewhere in the network. TFS incorporates a mechanism that detects these hot-spots and avoids using them for contribution.

3.1 Block Allocation

TFS ensures good file system performance by minimizing the amount of work that the file system performs when writing ordinary files. TFS simply treats transparent blocks as if they were free, overwriting whatever data might be currently stored in them. This policy allows block allocation for ordinary files to proceed exactly as it would if there were no transparent files present. This approach preserves performance for ordinary files, but corrupts data stored in transparent files. If an application were to read the transparent file after a block was overwritten, it would receive the data from the ordinary file in place of the data that had been overwritten. This presents two issues: applications using transparent files must ensure the correctness of all file data, and sensitive information stored in ordinary files must be protected from applications trying to read transparent files. To prevent both effects, TFS records which blocks have been overwritten so that it can avoid treating the data in those blocks as valid transparent file data. When TFS overwrites a transparent file, it marks it as *overwritten* and allocates the block to the ordinary file.

This requires some modifications to the allocation policy in Ext2. Blocks in a typical file system can only be in one of two states: *free* and *allocated*. In contrast, in TFS a storage block can be in one of five states: *free*, *allocated*, *transparent*, *free-and-overwritten*, and *allocated-and-overwritten*.

Figure 2 shows a state transition diagram for TFS blocks. Ordinary data can be written over free or transparent blocks. If the block was previously used by transparent data, the file system marks these blocks as allocated-and-overwritten. When a block is denoted as overwritten, it means that the transparent data has been overwritten, and thus "corrupted" at some point. Transparent data can only be written to free blocks. It cannot overwrite allocated blocks, other transparent blocks, or overwritten blocks of any sort. Figure 3 shows this from the perspective of the block map. Without TFS, appending to a file leads to fragmentation, leading to a performance loss in the write, and in subsequent reads. In TFS, the file remains contiguous, but the transparent file data is lost and the blocks are marked as *allocated-and-overwritten*.

When a process opens a transparent file, it must verify that none of the blocks have been overwritten since the last time it was opened. If any part of the file is overwritten, the file system returns an error to `open`. This signals that the file has been deleted. TFS then deletes the inode and directory entry for the file, and marks all of the blocks of the file as *free*, or *allocated*. As ordinary files cannot ever be overwritten, scanning the allocation bitmaps is not necessary when opening them. This lazy-delete scheme means that if TFS writes transparent files and never uses them again, the disk will eventually fill with overwritten blocks that could otherwise be used by the transparent storage application. To solve this, TFS employs a simple, user-space cleaner that opens and closes transparent files on disk. Any corrupted files will be detected and automatically deleted by the `open` operation.

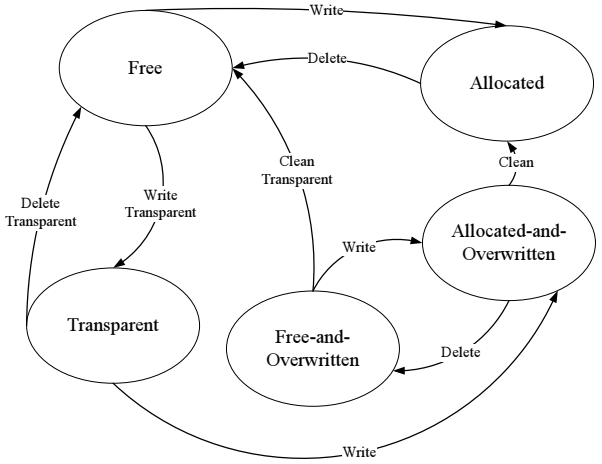

Figure 2: A state diagram for block allocation in TFS. The *Free* and *Allocated* states are the two allocation states present in the original file system. TFS adds three more states.

Though scanning the blocks is a linear operation in the size of the file, very little data must be read from the disk to scan even very large files. On a typical Ext2 file system, if we assume that file blocks are allocated contiguously, then scanning the blocks for a 4GB file will only require 384kB to be read from the disk. In the worst case—where the file is fragmented across *every* block group, and we must read every block bitmap—approximately 9.3MB will be read during the scan, assuming a 100GB disk.

Many file systems, including Ext2 and NTFS, denote a block's status using a bitmap. TFS augments this bitmap with two additional bitmaps and provides a total of three bits denoting one of the five states. In a 100GB file system with 4kB blocks, these bitmaps use only 6.25MB of additional disk space. These additional bitmaps must also be read into memory when manipulating files. However, very little of the disk will be actively manipulated at any one time, so the additional memory requirements are negligible.

3.2 Performance Concessions

This design leads to two issues: how TFS deals with open transparent files and how TFS stores transparent metadata. In each case, we make a small concession to transparent storage at the expense of ordinary file system per-

formance. While both concessions are strictly unnecessary, their negative impact on performance is negligible and their positive impact on transparent performance is substantial.

First, as TFS verifies that all blocks are clean only at open time, it prevents the file system from overwriting the data of open transparent files. One alternative would be to close the transparent file and kill the process with the open file descriptor if an overwritten block is detected. However, not only would it be difficult to trace blocks to file descriptors, but it could also lead to data corruption in the transparent process. In our opinion, yielding to open files is the best option. We discuss other strategies in Section 7.

It would also be possible to preserve the transparent data by implementing a copy-on-write scheme [26]. In this case, the ordinary file block would still allocate its target, and the transparent data block would be moved to another location. This is to ensure transparency with regards to the ordinary file allocation policy. However, to use this strategy, there must be a way to efficiently determine which inode the transparent data block belongs to, so that it can be relinked to point to the new block. In Ext2, and most other file systems, the file system does not keep a mapping from data blocks to inodes. Accordingly, using copy-on-write to preserve transparent data would require a scan of all inodes to determine the owner of the block, which would be prohibitively expensive. It is imaginable that a future file system would provide an efficient mapping from data blocks to inodes, which would allow TFS to make use of copy-on-write to preserve transparent data, but this conflicts with our goal of requiring minimal modifications to an existing operating system.

Second, TFS stores transparent meta-data such as inodes and indirect blocks as ordinary data, rather than transparent blocks. This will impact the usable space for ordinary files and cause some variation in ordinary block allocation decisions. However, consider what would happen if the transparent meta-data were overwritten. If the data included the root inode of a large amount of transparent data, all of that data would be lost and leave an even larger number of garbage blocks in the file system. Determining liveness typically requires a full tracing from the root as data blocks do not have reverse mappings to inodes and indirect blocks. Storing transparent storage metadata as ordinary blocks avoids both issues.

3.3 Transparent Data Allocation

As donated storage is constantly being overwritten by ordinary data, one concern is that constant deletion will have ill effects on any distributed storage system. Every time a file is deleted, the distributed system must detect and replicate that file, meanwhile returning errors to any peers that request it. To mitigate these effects, TFS identifies and avoids using the hot spots in the file system that could otherwise be used for donation. The total amount of space that is not used for donation depends on the bandwidth limits of the distributed system and is configurable, as shown in this section.

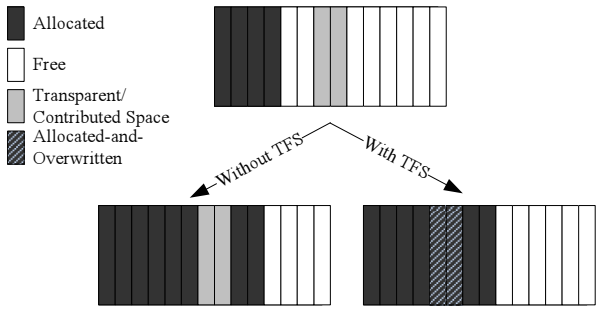

Figure 3: The block map in a system with and without TFS. When a file is appended in a normal file system it causes fragmentation, while in TFS, it yields two overwritten blocks.

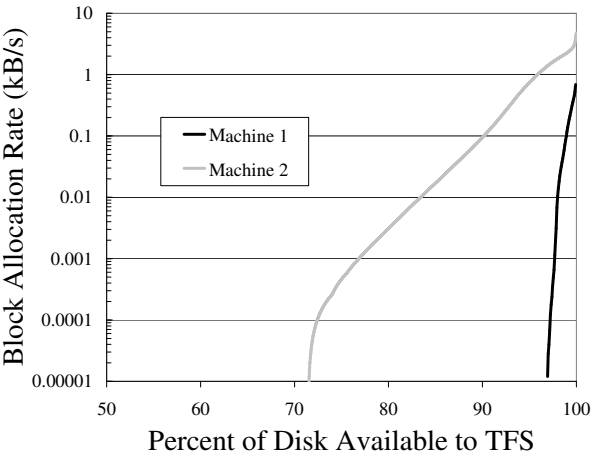

Figure 4: Cumulative histogram of two user machine's block allocations. 70% of the blocks on machine 2's disk were never allocated during the test period, and 90% of the blocks were allocated at a rate of 0.1kB/s or less. This corresponds to the rate at which TFS would overwrite transparent data if it were to use a given amount of the disk.

By design, the allocation policy for Ext2 and other logically organized file systems exhibits a high degree of spatial locality. Blocks tend to be allocated to only a small number of places on the disk, and are allocated repeatedly. To measure this effect, we modified a Linux kernel to record block allocations on two user workstations machines in our lab. A cumulative histogram of the two traces is shown in Figure 4. Machine 1 includes 27 trace days, and machine 2 includes 13 trace days. We can

observe two behaviors from this graph. First, while one user is a lot more active than the other, both show a great deal of locality in their access—machine 2 never allocated any blocks in 70% of the disk. Second, an average of 1kB/s of block allocations is approximately 84MB of allocations per day. Note that this is not the same as creating 84MB/s of data per day—the trace includes many short-lived allocations such as temporary lock files.

Using this observation as a starting point, TFS can balance the rate of block deletion with the usable storage on the disk. Using the same mechanism that we used to record the block allocation traces shown in Figure 4, TFS generates a trace of the block addresses of all ordinary file allocations. It maintains a histogram of the number of allocations that occurred in each block and periodically sorts the blocks by the number of allocations. Using this sorted list, it finds the smallest set of blocks responsible for a given fraction of the allocations.

The fraction of the allocations to avoid, f, affects the rate at which transparent data is overwritten. Increasing the value of f means fewer ordinary data allocations will overwrite transparent data. On the other hand, by decreasing the value of f, more storage becomes available to transparent data. Because the effects of changing f are dependent on a particular file system's usage pattern, we have found it convenient to set a target loss rate and allow TFS to determine automatically an appropriate value for f. Suppose ordinary data blocks are allocated at a rate of α blocks per second. If f is set to 0 – meaning that TFS determines that the entire disk is usable by transparent data – then transparent data will be overwritten at a rate approximately equal to α. The rate at which transparent data is overwritten t is approximately $\beta = (1 - f)\alpha$. Solving for f gives $f = 1 - \frac{\beta}{\alpha}$. Using this, TFS can determine the amount of storage available to transparent files, given a target rate. Using this map of hot blocks in the file system, the allocator for transparent blocks treats them as if they were already allocated to transparent data.

However, rather than tracking allocations block-by-block, we divide the disk into groups of disk blocks, called chunks, and track allocations to chunks. Each chunk is defined to be one eighth of a block group. This number was chosen so that each block group could keep a single byte as a bitmap representing which blocks should be avoided by transparent data. For the default block size of 4096 bytes, and the maximum block group size, each chunk would be 16MB.

Dividing the disk into multi-block chunks rather than considering blocks individually greatly reduces the memory and CPU requirements of maintaining the histogram, and due to spatial locality, a write to one block is a good predictor of writes to other blocks in the same chunk. This gives the histogram predictive power in avoiding hot blocks.

It should be noted that the rate at which transparent data is overwritten is not exactly α because, when a transparent data block is overwritten, an entire file is lost. However, because of the large chunk size and the high locality of chunk allocations, subsequent allocations for ordinary data tend to overwrite other blocks of the same transparent file, making the rate at which transparent data lost approximately equal to the rate at which blocks are overwritten.

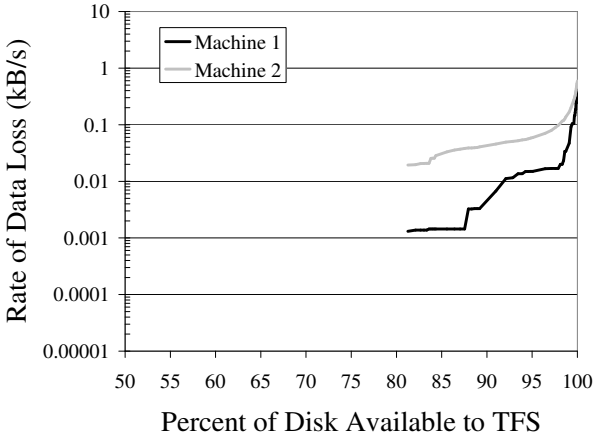

Figure 5: This shows the simulated rate of TFS data loss when using block avoidance to avoid hot spots on the disk. For example, when TFS allows 3% of the disk to go unused, the file system will allocate data in the used portion of the disk at a rate of 0.1kB/s. By using block avoidance, TFS provides more storage to contributory applications without increasing the rate at which data is lost.

The figure of merit is the rate of data erased for a reasonably high allocation of donated storage. To examine TFS under various allocations, we constructed a simulator using the block allocation traces used earlier in the section. The simulator processes the trace sequentially, and periodically picks a set of chunks to avoid. Whenever the simulator sees an allocation to a chunk which is not being avoided, it counts this as an overwrite. We ran this simulator with various fixed values of f, the fraction of blocks to avoid, and recorded the average amount of space contributed, and the rate of data being overwritten. Figure 5 graphs these results. Note that the graph starts at 80% utilization. This is dependent on the highest value for f we used in our test. In our simulation, this was 0.99999. Also notice that, for contributions less than approximately 85%, the simulated number of overwrites is greater than the number given in Figure 4. This is because, for very high values of f, the simulator's adaptive algorithm must choose between many chunks, none of which have received very many allocations. In this

case, it is prone to make mistakes, and may place transparent data in places that will see allocations in the near future. These results demonstrate that for machine 2's usage pattern, TFS can donate all but 3% of the disk, while only erasing contributed storage files at 0.08kB/s. As we demonstrate in the section 4, when compared to the replication traffic due to machine failures, this is negligible.

3.4 TFS Implementation

We have implemented a working prototype of TFS in the Linux kernel (2.6.13.4). Various versions of TFS have been used by one of the developers for over six months to store his home directory as ordinary data and Freenet data as transparent data.

TFS comprises an in-kernel file system and a user-space tool for designating files and directories as either transparent or opaque, called `setpri`. We implemented TFS using Ext2 as a starting point, adding or modifying about 600 lines of kernel code, in addition to approximately 300 lines of user-space code. The primary modifications we made to Ext2 were to augment the file system with additional bitmaps, and to change the block allocation to account for the states described in Section 3. Additionally, the `open` VFS call implements the lazy-delete system described in the design. In user-space, we modified several of the standard tools (including `mke2fs` and `fsck`) to use the additional bitmaps that TFS requires. We implemented the hot block avoidance histograms in user-space using a special interface to the kernel driver. This made implementation and experimentation somewhat easier; however, future versions will incorporate those functions into the kernel. An additional benefit is that the file system reports the amount of space available to ordinary files as the free space of this disk. This causes utilities such as `df`, which are used to determine disk utilization, to ignore transparent data. This addresses the concerns of users who may be worried that contributory applications are consuming their entire disk.

In our current implementation, the additional block bitmaps are stored next to the original bitmaps as file system metadata. This means that our implementation is not backwards-compatible with Ext2. However, if we moved the block bitmaps to Ext2 data blocks, we could create a completely backwards-compatible version, easing adoption. We believe that TFS could be incorporated into almost any file system, including Ext3 and NTFS.

4 Storage Capacity and Bandwidth

The usefulness of TFS depends on the characteristics of the distributed system it contributes to, including the dynamics of machine availability, the available bandwidth, and the quantity of available storage at each host. In this section, we show the relationship between these factors, and how they affect the amount of storage available to contributory systems. We define the storage contributed as a function of the available bandwidth, the uptime of hosts, and the rate at which hosts join and leave the network. Throughout the section we will be deriving equations which will be used in our evaluation.

4.1 Replication Degree

Many contributory storage systems use replication to ensure availability. However, replication limits the capacity of the storage system in two ways. First, by storing redundant copies of data on the network, there is less overall space [2]. Second, whenever data is lost, the system must create a new replica.

First we calculate the degree replication needed as a function of the average node uptime. We assume that nodes join and leave the network independently. Though this assumption is not true in the general case, it greatly simplifies the calculations here, and holds true in networks where the only downtime is a result of node failures, such as a corporate LAN. In a WAN where this assumption does not hold, these results still provide an approximation which can be used as insight towards the system's behavior. Mickens and Noble provide a more in-depth evaluation of availability in peer-to-peer networks [21].

To determine the number of replicas needed, we use a result from Blake and Rodrigues [3]. If the fraction of time each host was online is u, and each file is replicated r times, then the probability that no replicas of a file will be available at a particular time is

$$(1-u)^r. \tag{1}$$

To maintain an availability of a, the number of replicas must satisfy the equation

$$a = 1-(1-u)^r. \tag{2}$$

Solving for r gives the number of replicas needed.

$$r = \frac{ln(1-a)}{ln(1-u)} \tag{3}$$

We consider the desired availability a to be a fixed constant. A common rule of thumb is that "five nines" of availability, or $a = 0.99999$ is acceptable, and the value u is a characteristic of host uptime and downtime in the network. Replication could be accomplished by keeping complete copies of each file, in which case r would have to be an integer. Replication could also be implemented using a coding scheme that would allow non-integer values for r [28], and a different calculation for availability. In our analysis, we simply assume that r can take any value greater than 1.

4.2 Calculating the Replication Bandwidth

The second limiting effect in storage is the demand for replication bandwidth. As many contributory systems exhibit a high degree of *churn*, the effect of hosts frequently joining and leaving the network [13, 33], repairing failures can prevent a system from using all of its available storage [3]. When a host leaves the network for any reason, it is unknown when or if the host will return. Accordingly, all files which the host was storing must be replicated to another machine. For hosts that were storing a large volume of data, failure imposes a large bandwidth demand on the remaining machines. For instance, a failure of one host storing 100GB of data every 100 seconds imposes an aggregate bandwidth demand of 1GB/s across the remaining hosts. In this section, we consider the average bandwidth consumed by each node. When a node leaves the network, all of its data must be replicated. However, this data does not have to be replicated to a single node. By distributing the replication, the maximum bandwidth demanded by the network can be very close to the average.

The critical metric in determining the bandwidth required for a particular storage size is the *session time* of hosts in the network: the period starting when the host joins the network, and ending when its data must be replicated. This is not necessarily the same as the time a host is online—hosts frequently leave the network for a short interval before returning.

Suppose the average storage contribution of each host is c, and the average session time is T. During a host's session, it must download all of the data that it will store from other machines on the network. With a session time of T, and a storage contribution of c, the average downstream bandwidth used by the host is $B = \frac{c}{T}$. Because all data transfers occur within the contributory storage network, the average downstream bandwidth equals the average upstream bandwidth.

In addition to replication due to machine failures, both TFS and dynamic watermarking cause an additional burden due to the local erasure of files. If each host loses file data at a rate of F, then the total bandwidth needed for replication is

$$B = \frac{c}{T} + F. \tag{4}$$

Solving for the storage capacity as a function of bandwidth gives

$$c = T \cdot (B - F). \tag{5}$$

The file failure rate F in TFS is measurable using the methods of the previous section. The rate at which files are lost when contributing storage by the dynamic watermarking scheme is less than the rate of file loss with TFS. When using watermarking, this rate is directly tied to the rate at which the user creates new data.

If the value c is the total amount of storage contributed by each host in the network, then for a replication factor of r, the amount of unique storage contributed by each host is

$$C = \frac{c}{r} = \frac{T \cdot (B - F)}{r}. \tag{6}$$

The session time, T, is the time between when a host comes online and when its data must be replicated to another host, because it is going offline, or has been offline for a certain amount of time. By employing *lazy replication*—waiting for some threshold of time, t, before replicating its data—we can extend the average session time of the hosts [2]. However, lazy replication reduces the number of replicas of a file that are actually online at any given time, and thus increases the number of replicas needed to maintain availability. Thus, both T, the session time, and r, the degree of replication, are functions of t, this threshold time.

$$C = \frac{T(t)(B - F)}{r(t)} \tag{7}$$

The functions $T(t)$ and $r(t)$ are sensitive to the failure model of the network in question. For instance, in a corporate network, machine failures are rare, and session times are long. However, in an Internet-based contributory system, users frequently sign on for only a few hours at time.

5 TFS Evaluation

Our goal in evaluating TFS is to assess its utility for contributing storage to a peer-to-peer file system. We compare each method of storage contribution that we describe in Section 2 to determine how much storage can be contributed, and the effects on the user's application performance. We compare these based on several metrics: the amount of storage contributed, the effect on the block allocation policy, and the overall effect on local performance.

In scenarios that are highly dynamic and bandwidth-limited, static contribution yields as much storage capacity as any of the other three. If the network is more stable, and has more bandwidth, the dynamic scheme provides many times the storage of the static scheme; however, it does so at the detriment of local performance. When bandwidth is sufficient and the network is relatively stable, as in a corporate network, TFS provides 40% more storage than dynamic watermarking, with no impact on local performance. TFS always provides at least as much storage as the other schemes without impacting local performance.

5.1 Contributed Storage Capacity

To determine the amount of storage available to a contributory system, we conduct trace-based analyses using the block avoidance results from Section 3, the analysis of the availability trace outlined in Section 3.3, and the relationship between bandwidth and storage described in Section 4.2. From these, we use Equation 7 to determine the amount of storage that can be donated by each host in a network using each of the three methods of contribution.

We assume a network of identical hosts, each with 100GB disks that are 50% full, and we use the block traces for Machine 2 in Section 3.3 as this machine represents the worst-case of the two machines. Given a fixed rate of data loss caused by TFS, we determine the maximum amount of data that can be stored by TFS based on the data from Figure 5. We assume that the fixed contribution and the dynamic contribution methods cause no data loss. Though the dynamic contribution method does cause data loss as the user creates more data on the disk, the rate of data creation by users in the long term is low [4]. We assume that the amount of non-contributed storage being used on the disk is fixed at 50%. For fixed contribution, each host contributes 5% of the disk (5 GB), the dynamic system contributes 35% of the disk, and TFS contributes about 47%, leaving 3% free to account for block avoidance.

To determine the functions $T(t)$ and $r(t)$, we analyze availability traces gathered from two different types of networks which exhibit different failure behavior. These networks are the Microsoft corporate network [4] and Skype super-peers [14]. The traces contain a list of time intervals for which each host was online contiguously. To determine the session time for a given threshold t, we first combine intervals that were separated by less than t. We then use the average length of the remaining intervals and add the value t to it. The additional period t represents the time after a node leaves the network, but before the system decides to replicate its data.

We use these assumptions to calculate the amount of storage given to contributory systems with different amounts of bandwidth. For each amount of bandwidth, we find the value for the threshold time (Section 4.2) that maximizes the contributed storage for each combination of file system, availability trace, and bandwidth. We use this value to compute the amount of storage available using Equation 7. Figure 6 shows the bandwidth vs. storage curve using the reliability model based on availability traces of corporate workstations at Microsoft [4]. Figure 7 shows similar curves using the reliability model derived from traces of the Skype peer-to-peer Internet telephone network [14].

Each curve has two regions. In the first region, the total amount of storage is limited by the available bandwidth, and increases linearly as the bandwidth is increased. The slope of the first part of the curve is determined by the frequency of machine failures and file failures. This line is steeper in networks where hosts tend to be more reliable, because less bandwidth is needed to replicate a large amount of data. In this case, the amount of available storage does not affect the amount of usable storage. This means that, for the first part of the curve when the systems are bandwidth-limited, TFS contributes an amount of storage similar to the other two systems. Though TFS contributes slightly less because of file failures, the additional bandwidth needed to handle failures is small.

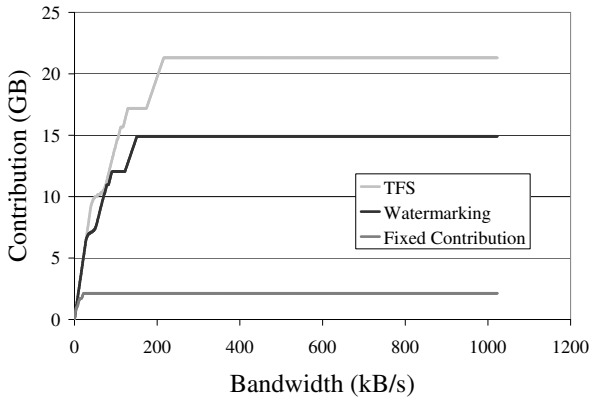

Figure 6: The amount of storage that can be donated for contributions of network bandwidth, assuming a corporate-like failure model. With reliable machines, TFS is able to contribute more storage than other systems, even when highly bandwidth-limited.

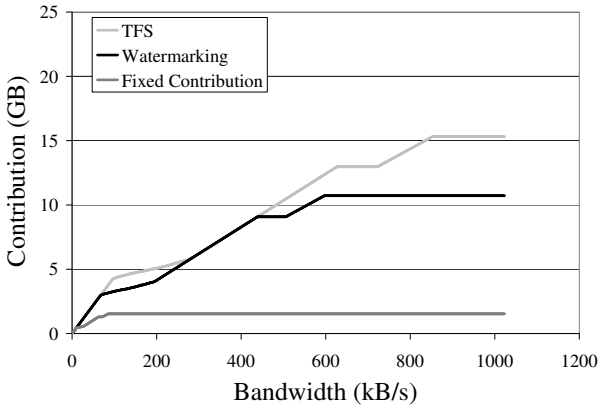

Figure 7: The amount of storage that can be donated for contributions of network bandwidth, assuming an Internet-like failure model. Because peer-to-peer nodes on the Internet are less reliable, the amount of contribution by TFS does not surpass the other techniques until large amounts of bandwidth become available.

The second part of the curve starts when the amount of storage reaches the maximum allowed by that contribution technique. For instance, when the small contribution reaches 5% of the disk, it flattens out. This part of the curve represents systems that have sufficient replication bandwidth to use all of the storage they are given, and are only limited by the amount of available storage. In this case, TFS is capable of contributing significantly more storage than other methods.

In the Microsoft trace, the corporate systems have a relatively high degree of reliability, so the bandwidth-limited portion of the curves is short. This high reliability means that, even for small bandwidth allocations, TFS is able to contribute the most storage. The Skype system shows a less reliable network of hosts. Much more network bandwidth is required before TFS is able to contribute more storage than the other storage techniques can—in fact, much more bandwidth than is typically available in Internet connected hosts. However, even when operating in a bandwidth-limited setting, TFS is able to contribute as much as the other techniques. One method to mitigate these bandwidth demands is to employ background file transfer techniques such as TCP-Nice [34].

From these results, we can conclude that TFS donates nearly as much storage as other methods in the worst case. However, TFS is most effective for networks of reliable machines, where it contributes 40% more storage than a dynamic watermarking system. It is important to note that these systems do not exhibit the same impact on *local* performance, which we demonstrate next.

5.2 Local File System Performance

To show the effects of each system on the user's file system performance, we conduct two similar experiments. In the first experiment, a disk is filled to 50% with ordinary file data. To achieve a realistic mix of file sizes, these files were taken from the `/usr` directory on a desktop workstation. These files represent the user's data and do not change during the course of the experiment. After this, files are added to the system to represent the contributed storage.

We considered four cases: no contribution, small contribution, large contribution, and TFS. The case where there is no simulated contribution is the baseline. Any decrease in performance from this baseline is interference caused by the contributed storage. The small contribution is 5% of the file system. This represents a fixed contribution where the amount of storage contributed must be set very small. The large contribution is 35% of the file system. This represents the case of dynamically managed contribution, where a large amount of storage can be donated. With TFS, the disk is filled completely with transparent data. Once the contributed storage is added, we run a version of the Modified Andrew Benchmark [25] to determine the contribution's effect on performance.

We perform all of the experiments using two identical Dell Optiplex SX280 systems with an Intel Pentium 4 3.4GHz CPU, 800MHz front side bus, 512MB of RAM, and a 160GB SATA 7200RPM disk with 8MB of cache. The trials were striped across the machines to account for any subtle differences in the hardware. We conduct ten trials of each experiment, rebooting between each trial, and present the average of the results. The error bars in all figures in this section represent the standard deviation of our measurements. In all cases, the standard deviation was less than 14% of the mean.

5.2.1 The Andrew Benchmark

The Andrew Benchmark [15] is designed to simulate the workload of a development workstation. Though most users do not compile programs frequently, the Andrew Benchmark can be viewed as a test of general small-file performance, which is relevant to all workstation file systems. The benchmark starts with a source tree located on the file system being tested. It proceeds in six phases: mkdir, copy, stat, read, compile, and delete.

Mkdir During the mkdir phase, the directory structure of the source tree is scanned, and recreated in another location on the file system being tested.

Copy The copy phase then copies all of the non-directory files from the original source tree to the newly created directory structure. This tests small file performance of the target file system, both in reading and writing.

Stat The stat phase then scans the newly created source tree and calls `stat` on every file.

Read The read phase simply reads all data created during the copy phase.

Compile The compile phase compiles the target program from the newly created source tree.

Delete The delete phase deletes the new source tree.

The Andrew Benchmark has been criticized for being an old benchmark, with results that are not meaningful to modern systems. It is argued that the workload being tested is not realistic for most users. Furthermore, the original Andrew Benchmark used a source tree which is too small to produce meaningful results on modern systems [15]. However, as we stated above, the Benchmark's emphasis on small file performance is still relevant to modern systems. We modified the Andrew Benchmark to use a Linux 2.6.14 source tree, which consists of 249MB of data in 19577 files. Unfortunately, even with this larger source tree, most of the data used by the benchmark can be kept in the operating system's page cache. The only phase where file system performance has a significant impact is the copy phase.

Despite these shortcomings, we have found that the results of the Andrew Benchmark clearly demonstrate the negative effects of contributing storage. Though the only significant difference between the contributing case and the non-contributing case is in the copy phase of the benchmark, we include all results for completeness.

The results of this first experiment are shown in Figure 8. The only system in which contribution causes any appreciable effect on the user's performance is the case of a large contribution with Ext2. Both the small contribution and TFS are nearly equal in performance to the case of no contribution.

It is interesting to note that the performance of TFS with 50% contribution is slightly better than the performance of Ext2 with 0% contribution. However, this does not show that TFS is generally faster than Ext2, but that for this particular benchmark TFS displays better performance. We suspect that this is an artifact of the way the block allocation strategy was be modified to accommodate TFS. As we show in Section 5.3, when running our Andrew Benchmark experiment, TFS tends to allocate the files used by the benchmark to a different part of the disk than Ext2, which gives TFS slightly better performance compared to Ext2. This is not indicative of a major change in the allocation strategy; Ext2 tries to allocate data blocks to the same block group as the inode they belong to [5]. In our Andrew Benchmark experiments, there are already many inodes owned by the transparent files, so the benchmark files are allocated to different inodes, and therefore different block groups.

The second experiment is designed to show the effects of file system aging [32] using these three systems. Smith and Seltzer have noted that aging effects can change the results of file system benchmarks, and that aged file systems provide a more realistic testbed for file system performance. Though our aging techniques are purely artificial, they capture the long term effects of continuously creating and deleting files. As contributory files are created and deleted, they are replaced by files which are often allocated to different places on the disk. The long term effect is that the free space of the disk becomes fragmented, and this fragmentation interferes with the block allocation algorithm. To simulate this effect, we ran an experiment very similar to the first. However, rather than simply adding files to represent contributed storage, we created and deleted contributory files at random, always staying within 5% of the target disk utilization. After repeating this 200 times, we proceeded to benchmark the file system.

Figure 9 shows the results of this experiment. As with the previous experiment, the only system that causes any interference with the user's applications is the large contribution with Ext2. In this case, Ext2 with 35% contribution takes almost 180 seconds to complete the benchmark. This is about 20 seconds longer than the same system without aging. This shows that file activity caused by contributory applications can have a noticeable impact on performance, even after considering the impact of the presence of those files. On the other hand, the time

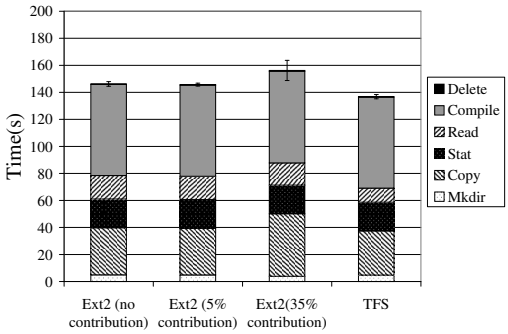

Figure 8: Andrew benchmark results for 4 different unaged file systems. The first is an Ext2 system with no contributory application. The second is Ext2 with a minimal amount of contribution (5%). The third has a significant contribution (35%). The fourth is TFS with complete contribution. TFS performance is on par with Ext2 with no contribution. Error bars represent standard deviation in total time.

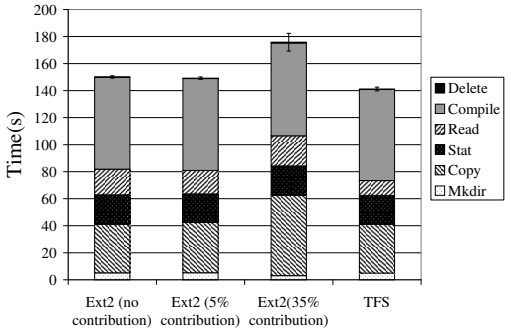

Figure 9: Andrew benchmark results for 4 different aged file systems. The first is an Ext2 system with no contributory application. The second is Ext2 with a minimal amount of contribution (5%). The third has a significant contribution (35%). The fourth is TFS with complete contribution. TFS performance is still comparable to Ext2 with no contribution. Error bars represent standard deviation in total time.

Figure 10: Block allocation pattern for several contribution levels in Ext2 and TFS. Both TFS and Ext2 with 0% contribution show high locality. Ext2 with 40% contribution does not because the contributed files interfere with the block allocation policy.

that TFS takes to complete the benchmark is unchanged by aging in the contributory files. There are no long-term effects of file activity by contributory systems in TFS.

5.3 Block Allocation Layout

A closer look at the block allocation layout reveals the cause of the performance difference between Ext2 and TFS. We analyzed the block layout of files in four occupied file systems. Two of these systems were using Ext2, the third was TFS. Each file system was filled to 50% capacity with ordinary data. Data was then added to simulate different amounts of data contribution. For Ext2, we consider two levels of contribution: 0% and 40%. 0% and 40% were chosen as examples of good and bad Ext2 performance from Figure 1. The TFS system was filled to 50% capacity with ordinary data, and the rest of the disk was filled with transparent files. We then copied the files that would be used in the Andrew Benchmark into these disks, and recorded which data blocks were allocated for the benchmark files.

Figure 10 shows the results of this experiment. The horizontal axis represents the location of the block on the disk. Every block that contains data to be used in the Andrew Benchmark is marked black. The remaining blocks are white, demonstrating the amount of fragmentation in the Andrew Benchmark files. Note that both Ext2 with 0% contribution, and TFS show very little fragmentation. However, the 40% case shows a high degree of fragmentation. This fragmentation is the primary cause of the performance difference between Ext2 with and without contribution in the other benchmarks.

6 Related Work

Our work brings together two areas of research: techniques to make use of the free space in file systems, and the study of churn in peer-to-peer networks.

Using Free Disk Space: Recognizing that the file system on a typical desktop is nearly half-empty, researchers have investigated ways to make use of the extra storage. FS^2 [16] is a file system that uses the extra space on the disk for block-level replication to reduce average seek time. FS^2 dynamically analyzes disk traffic to determine which blocks are frequently used together. It then creates replicas of blocks so that the spatial locality on disk matches the observed temporal locality. FS^2 uses a policy that deletes replicas on-demand as space is needed. We believe that it could benefit from a TFS-like allocation policy, where all replicas except the primary one would be stored as transparent blocks. In this way, the entire disk could be used for block replication.

A number of peer-to-peer storage systems have been proposed that make use of replication and free disk space to provide reliability. These include distributed hash tables such as Chord [24] and Pastry [29], as well as complete file systems like the Chord File System [9], and Past [30].

Churn in Peer-to-Peer Networks: The research community has also been active in studying the dynamic behavior of deployed peer-to-peer networks. Measurements of churn in live systems have been gathered and studied as well. Chu et al. studied the availability of nodes in the Napster and Gnutella networks [6]. The Bamboo DHT was designed as an architecture that can withstand high levels of churn [27]. The Bamboo DHT [23] is particularly concerned with using a minimum amount of "maintenance bandwidth" even under high levels of churn. We believe that these studies give a somewhat pessimistic estimate of the stability of future peer-to-peer networks. As machines become more stable and better connected, remain on continuously, and are pre-installed with stable peer-to-peer applications or middleware, the level of churn will greatly diminish, increasing the value of TFS.

7 Future Work

TFS was designed to require minimal support from contributory applications. The file semantics provided by TFS are no different than any ordinary file system. However, modifying the file semantics would provide opportunities for contributory applications to make better use of free space. For instance, if only a small number of blocks from a large file are overwritten, TFS will delete the entire file. One can imagine an implementation of TFS where the application would be able to recover the non-overwritten blocks. When a file with overwritten blocks is opened, the `open` system call could return a value indicating that the file was successfully opened, but some blocks may have been overwritten. The application can then use `ioctl` calls to determine which blocks have been overwritten. An attempt to read data from an overwritten block will fill the read buffer with zeros.

In the current implementation of TFS, once a transparent file is opened, its blocks cannot be overwritten until the file is closed. This allows applications to assume, as they normally do, that once a file is opened, reads and writes to the file will succeed. However, this means that transparent files may interfere with ordinary file activity. To prevent this, it would be possible to allow data from opened files to be overwritten. If an application attempts to read a block which has been overwritten, the read call could return an error indicating why the block is not available.

Both of these features could improve the service provided by TFS. By allowing applications to recover files that have been partially overwritten, the replication bandwidth needed by systems using TFS is reduced to the rate at which the user creates new data. By allowing open files to be overwritten, transparent applications may keep large files open for extended periods without impacting the performance of other applications. Despite these benefits, one of our design goals for TFS is that it should be usable by unmodified applications. Both of these features would require extensive support from contributory applications, violating this principle.

8 Conclusions

We have presented three methods for contributing disk space in peer-to-peer storage systems. We have included two user-space techniques, and a novel file system, TFS, specifically designed for contributory applications. We have demonstrated that the key benefit of TFS is that it leaves the allocation for local files intact, avoiding issues of fragmentation—TFS stores files such that they are completely transparent to local access. The design of TFS includes modifications to the free bitmaps and a method to avoid hot-spots on the disk.

We evaluated each of the file systems based the amount of contribution and its cost to the local user's performance. We quantified the unreliability of files in TFS and the amount of replication bandwidth that is needed to handle deleted files. We conclude that out of three techniques, TFS consistently provides at least as much storage with no detriment to local performance. When the network is relatively stable and adequate bandwidth is available, TFS provides 40% more storage over the best user-space technique. Further, TFS is completely transparent to the local user, while the user-space technique creates up to a 100% overhead on local performance. We believe that the key to encouraging contribution to peer-to-peer systems is removing the barriers to contribution, which is precisely the aim of TFS.

The source code for TFS is available at `http://prisms.cs.umass.edu/tcsm/`.

Acknowledgments

This material is based upon work supported by the National Science Foundation under CAREER Awards CNS-0447877 and CNS-0347339, and DUE 0416863. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

We would also like to thank our anonymous reviewers for their helpful comments. We would like to give special thanks to Erez Zadok for his careful shepherding.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symposium on Operating System Design and Implementation (OSDI 2002)*. USENIX Association, December 2002.

[2] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey Voelker. Total Recall: System support for automated availability management. In *Proceedings of the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, pages 73–86, San Jose, CA, May 2004.

[3] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, pages 1–6, Lihue, Hawaii, May 2003.

[4] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 34–43, New York, NY, USA, 2000. ACM Press.

[5] Remy Card, Thodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*. Laboratoire MASI — Institut Blaise Pascal and Massachussets Institute of Technology and University of Edinburgh, December 1994.

[6] Jacky Chu, Kevin Labonte, and Brian Neil Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of ITCom: Scala-*

bility and Traffic Control in IP Networks II Conference, July 2002.

[7] James Cipar, Mark D. Corner, and Emery D. Berger. Transparent contribution of memory. In *USENIX Annual Technical Conference (USENIX 2006)*, pages 109–114. USENIX, June 2006.

[8] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of Designing Privacy Enhancing Technologies: Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[10] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1999)*, pages 59–70, New York, NY, USA, 1999. ACM Press.

[11] Michael J. Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[12] `http://freenetproject.org/faq.html`.

[13] P. Brighten Goedfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. In *Proc. of ACM SIGCOMM*, 2006.

[14] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS '06)*, Santa Barbara, CA, February 2006.

[15] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanana, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[16] Hai Huang, Wanda Hung, and Kang G. Shin. FS^2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of 20th ACM Symposium on Operating System Principles*, October 2005.

[17] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. *Computational Genomics*. Horizon, 2002. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology.

[18] Ozgur Can Leonard, Jason Neigh, Erez Zadok, Jefferey Osborn, Ariye Shater, and Charles Wright. The design and implementation of Elastic Quotas. Technical Report CUCS-014-02, Columbia University, June 2002.

[19] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, pages 275–288, 2002.

[20] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[21] James Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems. In *Proceedings of the ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[22] Microsoft Corporation. `http://www.microsoft.com/technet/prodtechnol/winxppro/reskit/c28621675.mspx`.

[23] Ratul Mahajan Miguel. Controlling the cost of reliability in peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[24] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[25] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of USENIX Summer*, pages 247–256, 1990.

[26] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, May 2005.

[27] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. Technical Report UCB/CSD-03-1299, EECS Department, University of California, Berkeley, 2003.

[28] Rodrigo Rodrigues and Barbara Liskov. High availability in DHTs: Erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems*, February 2005.

[29] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for

large-scale peer-to-peer systems. In *Proceedings of Middleware*, November 2001.

[30] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th SOSP (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[31] Margo Seltzer, Keith Bostic, Marshall Kirt McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Winter USENIX Technical Conference*, January 1993.

[32] Keith Smith and Margo Seltzer. File system aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.

[33] Daniel Stutzbach and Reza Rejaie. Towards a better understanding of churn in peer-to-peer networks. Technical Report UO-CIS-TR-04-06, Department of Computer Science, University of Oregon, November 2004.

[34] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. TCP Nice: A mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, 2002.