Transparent Contribution of Storage and Memory

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

Many research projects have proposed *contributory* systems that utilize the significant free disk space, idle memory, and wasted CPU cycles found on end-user machines. These applications include peer-to-peer backup, large-scale distributed storage, and distributed computation such as signal processing and protein folding. While users are generally willing to give up unused CPU cycles, a variety of subtle factors conspire to deter participation in disk and memory-based contributory systems. First, users are often reluctant to give up disk space even though they aren't using it. Second, as contribution consumes available space, disk performance suffers. Finally, contributory applications pollute the machine's memory, forcing user pages to be evicted to disk. This paging can disrupt user activity for seconds or even minutes.

In this paper, we describe the design and implementation of two operating system mechanisms to support transparent contribution of storage and memory. A *transparent file system* (TFS) leaves the entire disk available to the user while allowing contributory storage applications to use *all* of the free space. We show that TFS's layout and allocation algorithms permit contributory storage while slowing the file system by no more than 4%. A *transparent memory manager* (TMM) controls memory usage by contributory applications, ensuring that users will not notice an increase in the miss rate of their applications. TMM is able to protect user pages such that page miss overhead is limited to 1.7%, while donating hundreds of megabytes of memory.

1 Introduction

A host of recent advances in connectivity, software, and hardware has given rise to *contributory systems* for donating unused resources to collections of cooperating hosts. The most prominent examples of deployed systems of this type are peer-to-peer file sharing applications that enable users to donate outgoing bandwidth and storage and receive bandwidth and storage in return. Other systems, such as Folding@home and SETI@home, donate excess CPU cycles to science. The research community has been even more ambitious, proposing systems that harness idle disk space to provide large-scale distributed storage like Palimpsest [15], PAST [16], CFS [4], IVY [14], and Pastiche [3]. All of these applications, be they primarily storage or processing, require a donation of idle memory as well as space, since contributory applications consume memory for mapped files, heap and stack, as well as the buffer cache.

Contributory systems depend on the participation of a large number of users to provide the maximum benefits of aggregated capacity and bandwidth, load balancing, replication, and geographic diversity. To maximize the number of users, such applications should ideally be completely *transparent* with respect to normal, or what we refer to as *opaque*, applications. Such opaque applications may be interactive, batch, or even background applications, but users prioritize opaque use over contributory applications. Unfortunately, contributory applications are not at all transparent, leading to significant barriers to widespread participation:

Loss of free disk space. Despite the fact that end-user hard drives are often half empty [6, 10], many users are reluctant to give it up. As anecdotal evidence, three of the Freenet FAQs express the implicit desire to donate less disk space. Even when users are not actively using most of their disk space, the feeling that this space might not actually be available can stifle participation.

Impaired file system performance. An unintended consequence of contributed storage is that it impairs overall file system performance. As a disk becomes full, block allocation performance worsens due to an increase in the number of seeks required for reads and writes. Throughput can drop by as much as 77% in a file system that is only 75% full versus an empty file system [20]—the more storage one contributes, the worse the problem becomes.

Memory pollution. Finally, contributing storage and processing leads to memory pollution, forcing the eviction of the user's pages to disk. The result is that users who leave their machines for a period of time can be forced to wait for seconds or even minutes while their applications are brought back into physical memory. In this paper, we show that this figure can grow to as high as 50% degradation after only a five minute break.

A number of traditional scheduling techniques and policies, such as proportional shares [23], can prevent only some kinds of interference from contributory services. For instance, if the owner is not actively using the machine, a contributory service can use all of the resources of the machine. When the user resumes work, the resources can be reallocated to give fewer resources to the contributory service. However, the time needed for reallocation directly translates into the acceptability of the system. The resource allocation of a network link or processor can be changed in microseconds, faster than any user can notice. However, storage allocation and memory allocation do not work well with the same reallocation strategies. Due to their reliance on relatively slow disks, allocation policies for storage and memory take a noticeable time to adapt. Swapping mapped pages to disk or deleting files can take minutes while the user waits.

To support the goal of transparency, we propose two mechanisms for controlling interference in end-user contributory storage and memory. The first is the *transparent file system* (TFS). TFS enables instantaneous reallocation of storage resources in a host machine while allowing users to contribute 100% of their free space with less than 4% performance impact on opaque file performance. The contributed storage is transient in that the user's machine may delete this hosted data at any time. Such a system depends on application-level mechanisms to replicate that data to other host machines. Because most contributory storage systems assume such unreliable storage already, they need little to no modification. The second mechanism we present is *transparent memory management* (TMM), which protects opaque applications from interference by transparent applications. TMM ensures that contributed memory is transparent: opaque performance is identical whether or not the user contributes memory.

TMM and TFS can be used together, or in isolation; each provides a distinct property that has widespread applicability to other transparent tasks. For instance, in disk-based web caches, distributed databases, and file systems, persistence is secondary to performance. For these applications, TFS may be used without TMM. Similarly, in applications such as P2P file sharing, serving performance is not essential while persistence is required.

These strategies are dynamic: they automatically adjust the allocation given to opaque and transparent applications. This is in sharp contrast with schemes like Resource Containers that statically partition resources [7]. Such static allocations suffer from two problems. First, a static allocation will typically waste resources due to the lack of statistical multiplexing. If one class is not using the resource, the other class should be able to use it, but static allocations prevent this. Second, it is unclear how users should choose an appropriate static allocation. We believe that the primary concern of users is the effect that running a service has on the performance of their computers. It is not normally possible for a lay user to determine how that translates into an allocation.

In this paper, we first describe the design of both of our mechanisms, TFS and TMM, and present a working implementation that runs in a modern operating system, Linux 2.6.13. Second, we present empirical results that show that TFS can donate 100% of the free space of the disk while impacting file performance by less than 4%. We next demonstrate that TMM is able to accurately estimate the user's true working set size, donate the remaining memory in the system, and limit the impact of increased page misses to 1.7%. Finally, we describe related work and conclude.

2 Design

In designing the two mechanisms that address all three of the barriers to participation described above, we followed these guiding principles:

Eliminate interference. The primary goal of TFS and TMM is to make hosted applications completely transparent with respect to user, or opaque, processes. In our design, we find that this is a largely attainable goal. However, we have found certain circumstances where it is reasonable to allow some interference. We make these concessions because they have a minimal negative impact on opaque performance, but a significant positive impact on transparent performance.

Minimize the interface. We believe that the smaller the API, the more likely it is that mechanisms will be adopted for a broad range of programs. Requiring code or compiler modifications violates this principle. To designate the use of TFS, a program only needs to place files in a particular directory or one of its subdirectories. No program modifications are necessary, except if the program cannot cope with non-persistent files. For the intended uses, we

believe that most programs already deal with this properly. To designate the use of TMM, the programmer only needs to designate the contributory application as a transparent program. All other programs are assumed to be opaque processes.

Additionally, there are two necessary elements that this paper does not address: small time-scale contention and security. This system does not focus on problems presented by small time-scale contention for resources such as the disk or write-behind cache. If both foreground and background activities contend for a resource simultaneously, this issue must be solved through judicious use of scheduling and rate control, not the mechanisms described here. Although simple CPU scheduling prioritization such as `nice` may limit these effects, a complete solution should include these mechanisms as well. TMM and TFS are also not designed to provide a secure sand-boxing tool for hosted applications. There are many ways that transparent applications can intentionally misbehave: they can consume more memory than they are profitably using, and they can make non-persistent files persistent. This system is instead meant for hosting well-mannered guests, while providing those applications with automatic resource management. Most scheduling and sand-boxing solutions should be complementary to the design presented here.

2.1 Transparent File System

Two of the barriers to contributory storage systems are the apparent loss of free space and interference with block allocation. We solve both of these problems though the use of a novel filesystem named the Transparent File System (TFS). In TFS, the block allocation for files follows the default mechanism as closely as possible. Any deviation from that policy results in the loss of opaque performance.

The cost of this insulation is persistence. When TFS allocates a block for opaque use, it treats free blocks and transparent blocks the same, possibly overwriting transparent data. Files marked transparent may thus be overwritten and deleted at any time. Fortunately, contributory services are already written to run on unreliable hosts and cleanly tolerate faults. For instance, deleting a file from a BitTorrent peer may cause it to send errors for a while, but the service will continue to run.

2.2 Block Allocation

In TFS, opaque file performance is protected by minimizing the amount of work that TFS must do when writing opaque files. To illustrate, one possible strategy would be to delete transparent files synchronously when TFS needed to overwrite them with opaque data. Clearly, this strategy would severely impact opaque performance. Instead, TFS simply marks transparent blocks that it needs as *dirty* and allocates them to opaque files. Marking the blocks as dirty ensures that TFS does not present corrupted data to transparent applications.

In TFS, a storage block can be in one of five states: *free*, *opaque-allocated*, *transparent-allocated*, *free-and-dirty*, and *dirty-and-opaque-allocated*. By contrast, blocks in a typical file system can only be in one of two states: *free* and *allocated*.

Figure 1 shows a state transition diagram for TFS blocks. Opaque data can be written over free or transparent blocks. If the block was previously allocated to transparent data, the filesystem marks these blocks as dirty-and-opaque-allocated. When a block is denoted as dirty, 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 opaque allocated blocks, other transparent blocks, or dirty blocks of any sort.

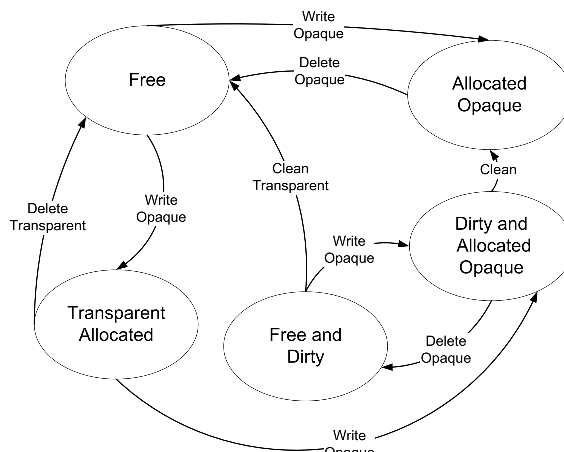

Figure 1:

When a process opens a transparent file, it must verify that none of the blocks have been dirtied since the last time it was opened. If any part of the file is dirty, the file system returns an error to open signaling that the file has been deleted, deletes the inode and directory entry for the file, and marks all of the blocks of the file as clean, or clean and opaque allocated. An alternate strategy would be to salvage as much of the file as possible by turning it into a sparse file. This strategy would require support from applications and run contrary to our design goal of a minimal API. The lazy delete scheme means that if TFS writes transparent files and never uses them again, the disk will eventually fill with dirty blocks that could otherwise be used by transparent processes. If

reclaiming the space is critical, TFS can employ a simple cleaner that simply opens and closes transparent files.

Many file systems, including Ext2 and NTFS, denote a block's status using a bitmap. TFS augments this bitmap with two additional bitmaps that provide three bits denoting one of the five states. In a 100GB file system with 512 byte blocks, these bitmaps use only 50MB 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, and the additional memory requirements are negligible.

2.2.1 Performance Concessions

This design leads to two questions: how should we deal with open transparent files, and how should we store transparent metadata? In each case, we make a small concession to transparent storage at the expense of opaque storage performance. While both concessions are strictly unnecessary, their negative impact on opaque performance is negligible and their positive impact on transparent performance is substantial.

First, TFS stores transparent metadata such as inodes and indirect blocks as opaque data. This transparent storage will impact the usable space and opaque block allocation. However, consider what would happen if the transparent metadata were overwritten. If the data included the root inode of a large amount of transparent data, all of that data would be lost. It may 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. This approach does create a possible security problem: a transparent process could store all of its data as metadata, e.g., in filenames and attributes. Since TFS does not delete such metadata, that storage would never be deleted. In keeping with our design goal, we ignore such security issues; however, simple limits can be placed on the total amount of metadata that a transparent process can create.

Second, TFS prevents opaque storage from overwriting the data of open transparent files. TFS can thus verify that all blocks are clean only at open time, yielding a form of session semantics. One alternative would be to close the file and kill the transparent process with the open file descriptor. However, not only would it be difficult to trace from blocks to file descriptors, it could also lead to data corruption in the transparent process. This policy raises the same security issue as the preservation of metadata described above, since it enables transparent processes to consume an arbitrary amount of disk space.

2.3 Transparent Memory Management

The third barrier to the acceptance of contributory systems is the pollution of the system's physical memory. As contributory applications read and write data from the file system, allocate heap space, use shared libraries, and memory map files, the physical memory of the machines will steadily fill. This active use of memory by the contributory application will eventually force the operating system to page the user's normal applications out. Opaque applications will then suffer from high memory access times. The problem is particularly acute whenever the user stops actively using the machine. The OS then assumes the opaque pages are inactive and evicts them. One alternative is to allocate no pages to the contributory application. Unfortunately, this defeats the point of contribution—users want to donate resources as long as it does not interfere with their opaque applications.

The goal of Transparent Memory Management (TMM) is to balance memory allocations between classes of applications. TMM allocates as much memory as it can to contributory applications, as long as that allocation is transparent to opaque applications. The insight is that opaque processes are often not using all of their pages profitably and can afford to donate some of them to contributory applications. The key is to decide how many pages to donate and to donate them without interfering with opaque applications. TMM automatically determines how many pages the opaque applications can afford to give up by maintaining a histogram of opaque page utilization. Using this histogram, TMM then determines how much memory opaque applications need, and thus what the allocation of memory should be to transparent applications.

Here we provide an overview of TMM: (i) it samples page accesses with a lightweight method using page reference bits, (ii) it uses these sampled references to create an approximate Least Recently Used (LRU) queue of the virtual memory, (iii) it uses the same sampled memory accesses and the LRU queue to create a histogram of memory accesses versus LRU queue position, (iv) it uses that histogram to determine how much memory the opaque applications are not using profitably, (v) it limits transparent memory usage to that amount, and (vi) it evicts opaque pages in an approximate LRU manner to free memory for transparent applications. We describe these in more detail below.

2.3.1 Determining Allocations

The key metric in memory allocation is the access time of virtual memory. As TMM donates memory to transparent tasks, opaque memory will be paged out, increasing the access time for opaque pages. Thus TMM determines the amount of memory to contribute by calculating what

that allocation will do to opaque page access times. The mean access time (MAT) for a memory page is determined from the hit ratio (h), the time to service a miss (m), and the time to service a hit (c):

$$\text{MAT} = (1 - h) \cdot m + h \cdot c. \quad (1)$$

If the system allows background processes to use opaque pages, it will increase the miss rate $(1 - h)$ of opaque applications by a factor of β, yielding an increase in MAT by a factor of:

$$\frac{\text{MAT'}}{\text{MAT}} = \frac{(1 - h) \cdot \beta \cdot m + (h \cdot \beta) \cdot c}{(1 - h) \cdot m + h \cdot c}. \quad (2)$$

In the case of a page miss, the page must be fetched from the page's backing store (either in the file system or from the virtual memory swap area), which takes a few milliseconds. On the other hand, a page hit is a simple memory access, and takes on average just a few tens of nanoseconds. Because these factors differ by many orders of magnitude, TMM can estimate that the average page access time is directly proportional to the number of page misses. Increasing the miss rate by β will make the ratio in (2) approximately β. This ratio is valid as long as there are some misses in the system. TMM uses a value of $\beta = 1.05$ by default, assuming that most users will not notice a 5% degradation in page access times.

If the user's working set is large, or there are no misses, TMM may determine that it cannot donate any pages to transparent processes. In this case, TMM makes a small concession to transparent applications, donating 256 pages (1MB). This amount provides a minimum level of performance that suffices for file read-ahead. Such a small allocation may cause contention for the disk head due to a high level of paging in transparent applications. To provide non-interference properties in this case, the disk scheduler must rate limit transparent swap activity. TMM does not yet prevent such small time-scale contention.

2.3.2 LRU Histogram

Limiting an allocation's effect to the factor β requires knowing the relationship between memory allocations and the miss rate. This relationship can be directly determined using a Least Recently Used (LRU) histogram, also known as a page-recency graph [19, 24, 25] or a stack distance histogram [1]. An LRU histogram allows TMM to estimate which pages will be in and out of core for any memory size. Using this histogram, TMM can determine what the miss rate for opaque processes will be if it allocates pages to transparent processes. Note that an LRU histogram uses an idealized view of an OS eviction policy—we explain how TMM manages page evictions later in this section.

In an LRU histogram H, the value at position x represents the number of accesses to position x of the queue. Thus $\sum_{i=0}^{x}[H(x)]$ is the number of accesses to all positions of the histogram up to and including x. This value is equivalent to the number of page hits that would have occurred in a system that had a memory size of x pages. Subtracting this value from the total number of accesses in the workload gives the number of misses for that given memory size. It is important to note that the LRU histogram contains all of the *virtual* pages in the system. With only the physical pages, it would not be able to predict what the miss rate would be given more memory pages. A sample cumulative histogram and memory allocation is shown in Figure 2.

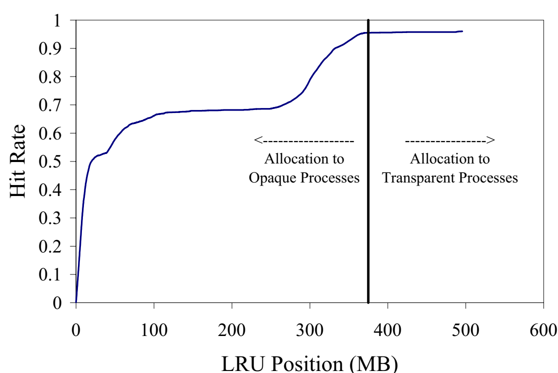

Figure 2: This figure demonstrates a sample histogram and allocation. In this case, the user has a working set size of approximately 390MB and can afford to donate the rest of the physical memory.

Building an exact histogram requires knowing the order of page references in the system and building an exact model of the LRU queue. When a page in memory is touched, its current position in the LRU queue is added to the histogram and the page is moved to the head. It is well-known that implementing a perfect LRU queue would severely degrade the performance of the machine, so TMM estimates the LRU queue based on a *sample* of page references. TMM maintains an LRU queue of virtual pages separate from the operating system's page eviction queues and keeps the pages in the queue in exact LRU order for any input of page accesses. For each access, TMM adds to the histogram the page's position in the LRU queue before the access was made, and then moves the page to the front of the queue.

It is important to note that determining a correct limit does not depend on a completely accurate histogram. We have assumed that allocations to transparent processes will be made so that opaque access has a very high hit rate, thus limiting interference. This implies that the only part of the histogram that needs to be accurate is the tail.

2.3.3 Sampling Page Hits

In order to build the LRU histogram, TMM needs a trace of memory accesses. One popular method in the memory management literature is to use a sampling approach triggered by page protections and faults [19, 24, 25]. These methods, while highly accurate, impose performance penalty on applications when handling page faults. Instead, TMM uses a lightweight sampling approach that leverages the MMU found in most modern hardware. The page reference bits provide an easy way of sampling page accesses without incurring the overhead of a page fault. TMM periodically clears a set of pages and then later checks their value. If the reference bit has been set, at least one access occurred in that period.

This sampling method raises several issues. First, sampling misses multiple accesses to the same page in a sampling interval and thus is an approximation of the real page access behavior. In TMM, the pages that are accessed more than once will typically be at the head of the histogram—the left-hand side of Figure 2—and missing references to the front will only make hits to the tail of the histogram look more significant. This is conservative as it will only increase the limit for opaque pages. Second, we conduct the sampling in a non-uniform manner. TMM favors accuracy at the tail of the LRU histogram and thus samples it more often.

2.3.4 Page Eviction

When applications allocate pages the operating system will first determine if there are free pages. If there are, it simply hands them to the process, regardless of the limits determined from the histograms—there is no reason to deny use of free pages. However, when free pages run low, the OS will force the eviction of other pages from the system and must choose between transparent and opaque pages. The choice depends on the limits determined by the factor β, and the current allocation of pages in the system. If both opaque and transparent applications are above their limits it favors opaque applications and evicts transparent pages. When evicting opaque pages, the performance of TMM heavily depends on evicting pages from the tail of the histogram.

However, it is well known that most OS do not use a perfect LRU eviction policy. Instead, many operating systems use an approximation of LRU called CLOCK [2]. We initially assumed that CLOCK would generally evict the pages that would produce the right performance guarantees—in principle, CLOCK should approximately evict pages from the tail of the LRU queue. However, after plotting the LRU queue position of page evictions, we discovered that this was not the case. In fact, CLOCK evicts pages from all over the LRU histogram. We first blamed this on the clearing of page reference bits for sampling. However, after making a duplicate of the page reference bit for the operating system to use, the problem remained.

Based on this observation, there are many ways to proceed. One option is to completely rewrite the eviction policy of the kernel to use our LRU estimates. This approach would be the best design, but would require a large amount of modifications to the kernel. Instead, we chose to massage CLOCK into favoring evictions from the tail of the LRU. By adding a hint to the page structure, we direct the CLOCK eviction policy to favor evictions of pages from the tail of the LRU queue. We use an integer between 0-19 to denote a quantized LRU queue position. This hinting is easily implemented and generally helps page eviction. However, better integration with the kernel would make this eviction procedure faster and much more accurate—we are currently pursuing this option. Figure 3 shows the difference in evicted LRU queue positions before and after adding this feature.

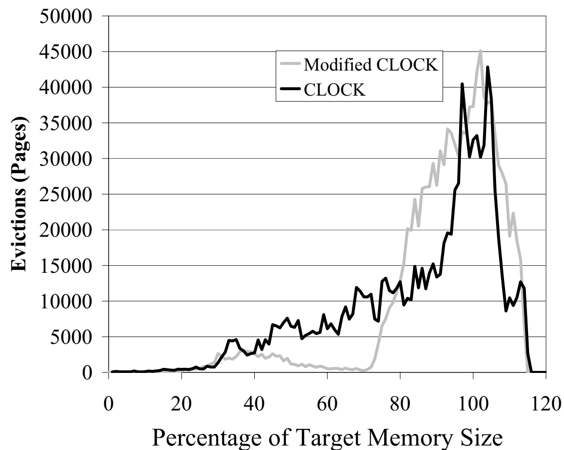

Figure 3: A histogram showing the LRU queue position of evicted pages. This graph compares Linux's default CLOCK eviction policy with TMM's eviction policy. TMM tends to evict the pages at, or past, a target memory size.

2.3.5 Aging the Histogram

The histogram that TMM uses to estimate opaque access patterns must adapt to changes in user behavior. An acceptable scheme must exhibit enough agility to adjust to changes in user behavior, but enough stability that it captures user activity over a long period of time.

We age the histogram over time using an exponentially weighted moving average. TMM keeps two histograms: a permanent histogram that TMM computes limits from, and a temporary histogram that records only the most re-

cent activity. After some amount of time t, we first divide the temporary histogram by the total number of accesses which happened in that aging period so that the values represent hit to miss ratios. We then add the temporary data into the permanent histogram using an exponential weighted moving average function and then recompute the cumulative histogram. We adopt a common value $\alpha = \frac{1}{16}$ and adjust the time t to match this.

The difficult part lies in tuning t. If TMM is not agile enough (too stable), rapid increases in opaque working set sizes will not be captured by the histogram, transparent applications will be allocated too many pages, and opaque page access times will suffer. If TMM is not stable enough (too agile), infrequently-used opaque applications will not register in the histogram and may be paged out.

To deal with this, we adopt a policy that is robust but favors opaque pages. Normally, t is set to 10 minutes for stability. This value is large enough that applications used in the last day or two bear enough significance in the histogram to force the memory limit to not page them out. However, to remain agile, the system must move the limit in response to unusually high miss rates. If TMM notices that the page misses have violated the stated goal, it adopts a more agile approach, using the most recent sample as the temporary histogram and immediately averaging it with the stable one. This policy has the disadvantage of stealing pages from transparent applications based on transient opaque use, but it is required to favor opaque applications over transparent ones.

To deal with startup effects we introduce two forms of aging: fast and slow. TMM uses fast aging when the machine first boots and set t=1 minute, while sampling page references 10 times per second. This yields higher CPU utilization, but "primes" the histogram with a starting value. After 15 minutes, TMM switches to its normal slow aging process with t=10 minutes and samples page references once per second.

2.3.6 Dealing with Noise

The goal in aging the histogram is to detect phase shifts in user activity over time. When the user is not actively using the machine, the histogram should remain static, directing TMM to preserve the opaque pages in the cache. However, we have observed that even when not actively using the machine, our Linux installation still incurs many page references from opaque applications. If left long enough, TMM misinterprets these accesses as shifts in user behavior and will change the allocation of pages to opaque applications. As there are only a small number of these pages, when the user is not using the system, it appears that the working set has very high locality. With very high locality, TMM will act improperly, allowing transparent allocations to consume a large number of pages in the system. In our Linux installation, this page activity comes from two primary sources: unattended cron jobs, and polling for changes in files.

Dealing with unattended cron jobs, such as locate index building, security scans, update management, and virus scans, is straightforward: we simply mark them as transparent processes. While such jobs are not contributory applications, the user benefits from limiting their memory consumption and their correspondingly reduced interference with other opaque applications. Polling for changes in files is becoming less of a problem with the advent of kernel-based upcall mechanisms for file monitoring, such as Linux's `inotify` and Microsoft Windows' `ChangeNotify`.

Nonetheless, both as a stopgap measure and to deal with poorly written programs, we choose to filter this background noise out of TMM. To decide whether or not to age the histogram after an aging period of t seconds, we only consider the opaque page accesses that have touched the LRU queue at a point farther than 10% of the total. If there were more than ten such accesses per second, we age the histogram. This policy also implies that we need to guarantee a minimum of 10% of the system's memory to opaque memory, a reasonable assumption. We found that the idle opaque activity rarely touches pages beyond the 10% threshold. We observed that with filtering, TMM never ages the histogram during periods of disuse.

3 Implementation

To demonstrate the efficacy of our approach, and to provide a test platform for our work, we have implemented complete and fully usable versions of both TFS and TMM in using a Linux 2.6.13.4 kernel. [1]. Various versions of TFS have been used by one of the developers for over six months to store his home directory as opaque data and Freenet data as transparent data.

TFS requires both an in-kernel file system and a user-space tool for designating files and directories as transparent or opaque, called `setpri`. We implemented TFS using Ext2 as the base file system. 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 2.1. 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

[1]The source code for TFS and TMM, including kernel modifications and user-space tools, can be downloaded from the following URL : *http://www.prisms.cs.umass.edu/TCSM/tcsm.tar.gz*

bitmaps that TFS requires. Unfortunately, due to the additional bitmaps, our implementation is not backwards-compatible with Ext2.

TMM requires a number of kernel modifications as well as several user-space tools. Inside the kernel, we implemented tracing methods that periodically mark pages as unreferenced, and then later test them for MMU-marked references. The list of page references and recent evictions are passed through a `/dev` interface. A user space tool written in C++ reads these values and tracks the LRU queue. This tool computes the transparent and opaque limits and passes that data back into the kernel through the device interface.

Additionally, we have augmented the Linux eviction policy kernel with our LRU-directed policy. As the LRU simulator runs in user space, we built a second approximate LRU queue into the kernel to direct which pages to sample and to provide the hints for the eviction policy. We plan to rewrite this part of the kernel, as it is the largest source of error in TMM. Another user-space tool, `maketransparent`, allows users to mark processes as transparent. All pages that the transparent processes use are then limited using TMM. Pages that are shared between opaque and transparent pages are marked as opaque pages, but to ignore noise caused by transparent process access, hits to those pages do not age the histogram.

4 Evaluation

In evaluating TFS and TMM, we sought to answer the following questions:

- What is the basic overhead of TFS? What is the performance impact of overwriting transparent data?
- Under a scenario where the file system is in use, how well does TFS prevent background allocation and fragmentation from affecting opaque file performance?
- How well does TMM prevent transparent processes from paging opaque memory out, and how does TMM's dynamic technique compare to static allocation?
- What is the transient performance of TMM?
- What is the overhead in using TMM?

To answer those questions, we used a series of file system and memory benchmarks that are designed to simulate typical use of an end-user system. For file system benchmarks, we used a modified version of the Andrew Benchmark [9] using a minimal configuration of the Linux 2.6.14.5 source tree. The source is 325 MB uncompiled and 347 MB compiled. While there may be nothing typical about the Andrew Benchmark, it does show a breakdown of individual file operations and provides a basis of comparison with other file systems. We use an unmodified Linux Ext2 file system as a point of comparison to TFS. For the TMM experiments, we used a variety of user applications, Mozilla, OpenOffice, KView, and Gnuplot to simulate typical use. Each experiment was conducted three times, with the exception of the transient experiments and the slow aging experiments.

We conducted all of the experiments using a small cluster of 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. In the case of TFS, we restricted our experiments to a 10GB partition to speed experimentation.

4.1 Transparent File System

Our first experiment demonstrates basic, opaque TFS performance using the Andrew Benchmark. In this experiment, the Andrew Benchmark was run on three file system configurations. The first was an ordinary, empty Ext2 file system. The other two were TFS file systems, one empty, and one filled to 100% with transparent files. In the two TFS file systems, the Andrew Benchmark was run as opaque data. The results of this experiment are shown in Figure 4.

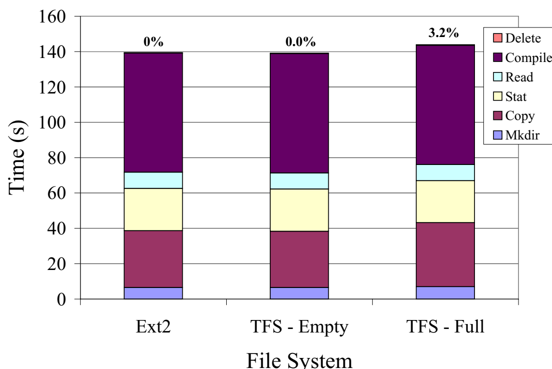

Figure 4: Andrew Benchmark tests comparing Ext2, TFS, and a TFS file system completely filled with low-priority data. An empty TFS system imposes no measurable overhead, and a full TFS system imposes less than 4%.

The second bar shows that with no contributed storage TFS imposes unmeasurable overhead over a basic Ext2 file system. More importantly, the third bar shows the

performance impact of the presence of transparent file data. This demonstrates that opaque file access is impeded very little by the presence of transparent data, thus meeting our goal of transparency. The 3.2% overhead is due to the extra block bitmaps on the disk, and extra block allocation logic inherent to TFS.

Empty file system performance establishes the baseline overhead of using TFS, but does not adequately show how real-world use affects performance. As a more realistic test, we compare Ext2 and TFS under a simulated contributory system. To provide the abstract properties of such a contributory application, we developed a synthetic benchmarking tool called SYNCFS. SYNCFS copies data to the free space of the host machine's disk. When the disk becomes full, SYNCFS deletes random parts of the downloaded data to return the file system to 15% free and repeats. SYNCFS replicates a mix of file sizes takes from a Linux machine's `/usr` directory.

First, we assume that the user has filled the disk to 55% full—according to one measurement study this is the median space used in end-user disks [6]. We then compare three different scenarios: (i) the user does not contribute any storage and uses Ext2, (ii) the user contributes storage using Ext2, (iii) the user contributes storage using TFS. When simulating contribution, we ran SYNCFS for 10 cycles of create-delete on the target file system. We then run the Andrew Benchmark and measure its performance. The results of using the three systems are shown in Figure 5.

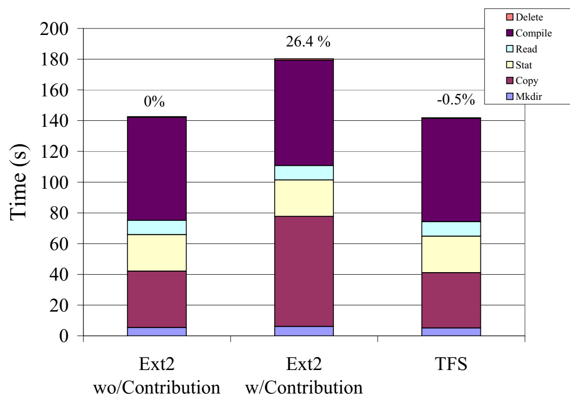

Figure 5: Andrew Benchmark tests comparing the effect of contribution. When contributing Ext2 and TFS perform similarly. When the contribution is handled by TFS, the file system performs the same as not contributing.

The first thing to note is the comparison between the the first and second bars. This difference shows the effect of contributing space on Ext2 performance. Most of the overhead comes in the copy phase of the benchmark—in the 85% full system writes are *100% slower*. The loss in performance is due to two factors. The first is that when using SYNCFS, the file system is now 85% full not 55% full, stressing the block allocator and lowering performance. The second is the effects of fragmentation, or aging, of the file system. Both of these factors have been demonstrated in previous work [20, 18]. When comparing the third bar to the other two, we see that TFS is able to mask both of these effects and provide the same performance as if the contributory application was not running. Note that the negative value is an artifact of experimental noise. It may be possible to remove the effects of aging by reorganizing or defragmenting the disk, but this is problematic as the disk is actually full, making defragmentation difficult or impossible. However, defragmentation does nothing to remove the performance impact of having a very full disk, which is the primary bottleneck.

4.2 Transparent Memory Management

The primary function of TMM is to ensure that contributory processes that use memory do not interfere with the user's applications. To show TMM's benefits, we simulate typical user behavior: we use several applications, take a coffee break for five minutes, and return to using similar applications. During the coffee break, the machines runs a contributory application. We use a program called POV-Ray, a widely-used distributed rendering application. For this experiment, we compare five systems with three different sets of opaque applications. The five systems are as follows: vanilla Linux, TMM with three different static allocations for opaque applications (25%, 50%, 75%), and TMM using its histogram-based limiting method. For vanilla Linux, we present results with and without the contributory application. The three different sets of applications represent different user activities with different working set sizes: Small (Mozilla), Medium (Mozilla, OpenOffice, and KView), and Large (Gnuplot with very large data set). We track the average and maximum number of page misses per second recorded in the first minute after the user comes back from break and present the results in Figures 6 and 7. Note that we rebooted between each trial, and the TMM results are based on our fast aging system presented in Section 2.3.5.

The first thing to note is that with a vanilla Linux kernel, the system running the contributory application performs very poorly, incurring as many as 190 page faults/sec on average. Assuming that the application is page fault limited, and given an average miss latency of 2.5ms, these faults cause a 50% slow down in the execution of the application. Qualitatively, we have observed that this is highly disruptive to the user. Second,

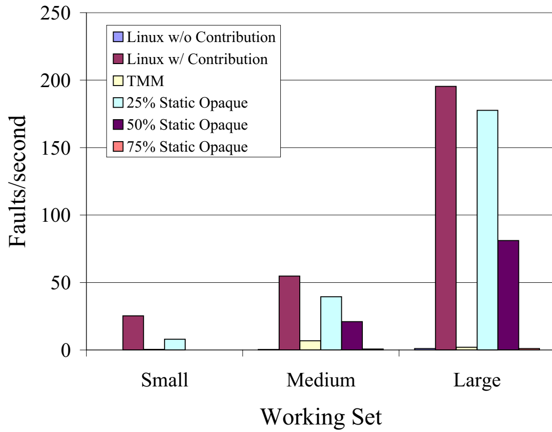

Figure 6: This figure shows the average page faults/sec in the first minute after resuming work. TMM performs much better than an unmodified operating system, and better than static limits, except when the static limit is set very high.

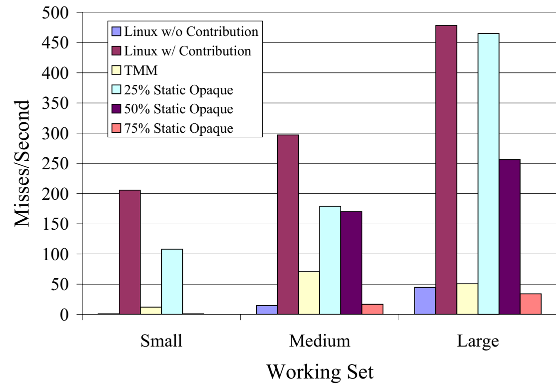

Figure 7: This figure shows the maximum page faults/sec of any second in the first minute after resuming work. Short term violations of the target 5% slowdown are possible, but TMM perfroms better than unmodified Linux and the static schemes.

for each static limit, there is a workload that performs worse than TMM, except for the 75% limit. In that case, the static limit performs well. However, we could have easily constructed a working set size larger than 75%, and TMM would have produced far fewer page misses than the static allocations.

Most importantly, the performance of TMM is comparable to the performance of the vanilla Linux system without contribution. The largest number of average page faults that TMM incurs is for the medium size working set, at 6.8 page faults/sec. By the same calculation used above, this faulting rate causes a 1.7% slow down in the opaque applications, well within our goal of 5%. This demonstrates TMM's ability to donate memory transparently. The few pages that TMM does evict could be recovered by making Linux's page eviction more LRU-like. As shown in Figure 7, short-term violations of our goal are possible—TMM statistically guarantees average performance over long periods. Nonetheless, TMM provides better short-term performance than the static limits (except 75%), and much better performance than unmodified Linux. For the large working set, even Linux without contribution does incur some page faults.

To measure the actual amount of memory that TMM donates to transparent applications, we ran the interactive phase of our benchmarks until it reached the slow aging stage, waited for the limit to stabilize, and recorded the transparent limits. We also recorded the amount of memory donated under the static limits. The static limits apply to the limit on opaque memory and thus transparent memory contribution also depends on consumption by the operating system. The results are presented in Figure 8.

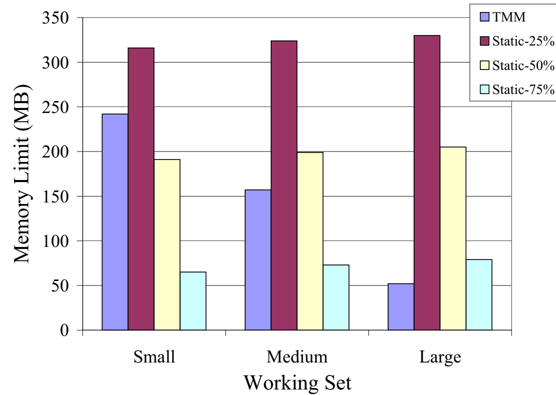

Figure 8: This figure shows the amount of memory in MB donated to transparent processes.

When comparing the amount of memory donated by each system, we show two things. First, TMM is conservative in the amount of memory it donates, favoring opaque page performance over producing a tight limit. Also, for static limits, we found that the user's working set does not necessarily indicate how much space the operating system uses. Nonetheless, for each static limit, there is a working set for which the limit fails to both preserve performance and contribute the maximum amount of memory. TMM succeeds in achieving both of these goals for every working set.

The next experiment demonstrates how TMM behaves over time. We conduct an experiment similar to the previous one and graph the memory use and memory limits that TMM sets. A timeline is shown in Figure 9. At the beginning of the timeline, the set of opaque processes

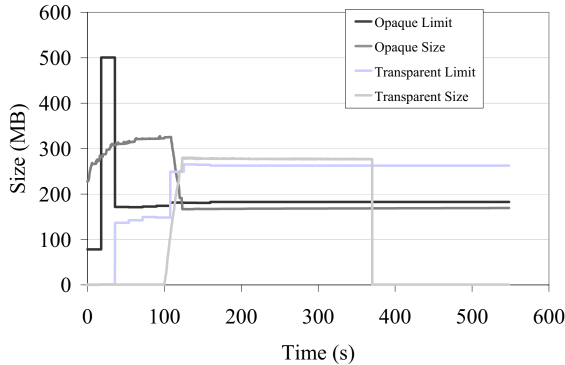

Figure 9: This figure shows a sample timeline of limits and utilization of the TMM system.

is using 320 MB of memory and there are no transparent applications in the system. TMM has set a limit for both opaque and transparent processes, but as there is no memory pressure in the system, the memory manager lets opaque processes use more memory than the limit. Note that at this time, the sum of the transparent and opaque limits is far less than the physical memory of the machine (512 MB), the rest of the memory is in free pages. At time 30, we start a transparent process that quickly consumes a large amount of the free memory. TMM now sees a larger pool to divide and increases the transparent limit. TMM does not adjust the opaque limit as the user has not changed behavior and does not need any more memory than the limit allows. The transparent process is now causing memory pressure in the system, forcing pages out. The number of opaque is over its limit so it loses them to the evictor. The graph exhibits some steady state error. We have tracked this to the zoned memory system that Linux uses, something we would correct in a redesigned evictor.

Lastly, it is important to note the CPU and memory overhead of TMM. TMM consumes approximately 6% of the CPU while in fast aging mode and less than 2% while in slow aging mode. This CPU time is primarily due to running TMM in user space requiring many user-kernel crossings to exchange page references and limits. TMM uses approximately 64MB of memory. This large memory overhead is due to duplicating the LRU list in the kernel, another straightforward optimization. A kernel implementation of TMM will reduce both of these overheads significantly.

5 Related Work

TFS and TMM bring together research from several diverse fields, such as support for transparency, use of free disk space, and memory management. Here we cover the most relevant work to our own.

5.1 Support for Transparency

A number of systems have been proposed to support resource management similar to our notion of transparency. For instance, TCP-Nice [22] is a facility to prevent certain network flows from interfering with higher priority network flows. Many of the applications that TCP Nice targets are the same applications that will use TFS, prefetching, content distribution, and peer-to-peer storage. TCP Nice is a necessary part of any complete system for supporting contributory services and we are currently working to integrate the two systems.

MS Manners [5] is a resource-general system for preventing low priority processes from interfering with high priority processes. Specifically it targets low priority applications such as a duplicate file merger and high priority processes such as SQL server. Several factors make MS Manners inappropriate for providing transparent memory allocations: First, it only applies to what they term *symmetrical* resources that degrade applications equally when there is contention. Memory contention doesn't degrade applications such as BitTorrent at all, whereas it has dire consequences for opaque applications such as a browser—the authors admit this shortcoming, saying that the system does not work for physical memory. Second, MS Manners is a reactive system, modifying allocations in response to measured reductions in application progress. Unlike highly loaded servers such as MS SQL, typical desktop applications are very bursty and have unknown progress rates.

Other facilities include the multitude of scheduling algorithms for manually setting shares and priorities for resources. UNIX facilities, such as `nice`, can be used to deprioritize processes access to the CPU, but cannot control access to other resources. Systems such as Lottery Scheduling [23] claim general applicability to resources, but are only appropriate for resources that are quickly renewed, such as bandwidth or CPU time, and can be quickly reallocated between contending parties. Similarly, Resource Containers apply general, but static, allocations to resources [7]. As mentioned in the introduction, such static allocations are inherently inefficient and do not determine what to set the allocations to. We have shown in our evaluation the benefits of using a dynamic scheme, such as TMM, over static allocations.

5.2 Free Disk Space

Many researchers have noted the large amount of free disk space, either locally or in aggregate, and have attempted to put that idle space to work. Elastic Quotas allow users to mark files that temporarily exceed their disk space quotas on multiuser machines [11]. A background daemon, named *rubberd*, defines a high and

low watermark for free space before eliminating these marked files. While Elastic Quotas support the goal of transparency, TFS has the distinct advantage in eliminating the watermarks. TFS also avoids interference with the block allocator, preserving performance for opaque files, and allowing transparent storage to use 100% of the empty space on the disk. In the evaluation, we have shown that creating a deleting files in the empty space causes significant performance penalties for opaque file access.

Most recently, FS^2 proposed using this disk space to provide replication of local data for performance and energy benefits [10]. FS^2 uses high and low watermarks for files, similar to Elastic Quotas, in order to avoid the same interference caused with the block allocator. We believe that FS^2 could also benefit from TFS' ability to use 100% of the space without the user noticing.

The Elephant file system proposed using the extra free space for keeping older versions of files for automatic versioning support [17], while S4 uses the extra space for security logging [21]. Elephant and FS^2 are example of other ways to use free space but do not address the performance issues that TFS solves.

5.3 Memory Management

Many efforts have been made to improve upon Least Recently Used replacement policies by incorporating the notion of frequency, including the recent work on ARC [13]. ARC uses two concurrent LRU queues to differentiate between pages that have been visited once recently and those that have been visited more than once, thus separating pages with high and low temporal locality. This helps to make ARC *scan resistant*, safely ignoring pages that are sequentially read, and avoiding page-cache pollution. Unfortunately, tuning the balance between recency and frequency will eventually fail to keep certain pages transparent. Without the OS knowing the difference between opaque and transparent activity, the OS will eventually believe that the transparent activity is the only activity on the machine and evict opaque pages. More importantly, there is no assumption that transparent pages managed by TMM have low temporal locality, and many contributory storage systems may in fact have high degrees of temporal locality.

The CacheCOW system [8] generally defines the problem of providing QoS in a buffer cache. CacheCOW contains some elements of TMM, including providing different hit rates to different classes of applications, but targets Internet servers in a theoretic and simulation context. CacheCOW does not address many of the practical issues in using, gathering, and aging LRU histograms.

LRU histograms [25], or page recency-reference graphs [24, 19], are useful in many contexts such as memory allocation and virtual memory compression and are essential to TMM. Some optimizations to gathering histograms have been implemented that rely on page protection but lower the overhead to 7-10% [26]. In this paper, we use an adapted form of tracing that requires less implementation effort in the kernel and avoids the overhead of handling relatively expensive page faults. The disadvantage of our referenced bit approach is less accuracy—the reference bit does not reveal multiple hits to the same page within one sampling period; however, we have found the accuracy sufficient given the requirements of TMM. Nonetheless, TMM is agnostic with respect to how the LRU histograms are obtained, as long as the overhead is acceptable.

5.4 Linux Support for Transparency

As we have implemented TFS and TMM on top of Linux, we note that there is some minimal support for non-interference in the 2.6 kernel. Specifically, Linux makes a distinction between two kinds of memory pages: memory mapped pages, and non-mapped pages. Memory mapped pages include `mmapped` files including shared libraries, and program heap, stack, and data space. Non-mapped pages include those added to the cache from regular file access. As part of its eviction policy, Linux 2.6 measures the ratio of mapped to unmapped pages and the level of *distress* in freeing pages. Using a unitless parameter called *swappiness*, a systems administrator can control contention between mapped and unmapped pages. Given the default value of swappiness, Linux generally prevents the allocation of non-mapped pages from forcing the eviction of mapped pages until the allocation of mapped pages exceeds 80% of physical memory. The consequence is that on many systems, a typical user may not use more than 80% mapped pages, and non-mapped accesses becomes transparent. Thus, contributory applications that only use non-mapped pages will not interfere with mapped opaque pages. For example, BitTorrent will not interfere with the user interface. In effect, Linux has inferred the importance of memory, and thus applications, using the type of memory the application used.

While this inference works in certain cases, it does not work in general. For instance, contributory applications using non-mapped pages will interfere with foreground non-mapped pages, such as those used by compilation or file management. Further, contributory applications that use mapped pages, such as simulations or distributed data-set processing, will interfere with opaque applications, such as X. Anecdotally, we have seen this in our lab's use of Condor [12]. After watching X laboriously page after coffee breaks, all of our users disabled Condor. To be fair, we have not disabled this inference feature in

Linux, but TMM will work slightly better when not competing with this kernel feature.

6 Conclusions

In this paper we present two operating system extensions, the Transparent File System (TFS) and the Transparent Memory Manager (TMM) for supporting transparent contribution of storage and memory. These systems prevent contributory applications from interfering with the performance of a user's application, while maximizing the benefits of harnessing idle resources. TFS is effective in preserving the performance of a user's file system, while still donating 100% of the free space on the disk. TMM protects the user's pages from unwarranted eviction and limits the impact on the performance of user applications to less than 5%, while donating hundreds of megabytes of idle memory.

References

[1] G. Almasi, C. Cascaval, and D. A. Padua. Calculating stack distances efficiently. In *The International Symposium on Memory Management (ISMM 2002)*, Berlin, Germany, June 2002.

[2] F. J. Corbato. *In Honor of Philip M. Morse*, chapter A Paging Experiment with the MULTICS System, pages 217–228. M.I.T. Press, 1968.

[3] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.

[4] 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.

[5] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of 17th ACM Symposium on Operating Systems Principles*, pages 247–60, December 1999.

[6] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 59–70, New York, NY, USA, 1999. ACM Press.

[7] P. Druschel G. Banga and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999.

[8] P. Goyal, D. Jadav, D. Modha, and R. Tewari. Cachecow: Qos for storage system caches. In *International Workshop on Quality of Service (IWQoS)*, Monterey, CA, June 2003.

[9] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[10] H. Huang, W. Hung, and K. 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.

[11] O. C. Leonard, J. Neigh, E. Zadok, J. Osborn, A. Shater, and C. Wright. The design and implementation of elastic quotas. Technical Report CUCS-014-02, Columbia University, June 2002.

[12] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[13] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of 2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003.

[14] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of the Fifth Symposium on Operating System Design and Implementation (OSDI 2002*, Boston, MA, December 2002.

[15] Timothy Roscoe and Steven Hand. Palimpsest: Soft-capacity storage for planetary-scale services. In Michael B. Jones, editor, *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, May 18-21, 2003, Lihue (Kauai), Hawaii, USA*. USENIX, 2003.

[16] A. Rowstron and P. 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.

[17] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 110–123, New York, NY, USA, 1999. ACM Press.

[18] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 Winter USENIX Technical Conference*, pages 249–264, New Orleans, LA, January 1995.

[19] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2), July 2003.

[20] K. Smith and M. Seltzer. File system aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.

[21] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the*

4th Symposium on Operating Systems Design and Implementation (OSDI), pages 165–179, San Diego, CA, October 2000.

[22] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A mechanism for background transfers. In *Proceedings of the Fifth USENIX Operating System Design and Implementation*, December 2002.

[23] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, November 1994.

[24] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116, Monterey, CA, June 1999.

[25] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the Third International Symposium on Memory Management (ISMM)*, Vancouver, October 2004.

[26] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamically tracking miss-ratio-curve for memory management. In *The Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.