Peabody: The Time Travelling Disk

Charles B. Morrey III
Univ. of Colorado, Boulder
cbmorrey@cs.colorado.edu

Dirk Grunwald
Univ. of Colorado, Boulder
grunwald@cs.colorado.edu

Abstract

Disk drives are now available with capacities on the order of hundreds of gigabytes. What has not become available is an easy way to manage storage. With installed machines located across the enterprise, the backup, management of application installation, and maintenance of systems have become a nightmare. An increasing trend in the storage industry is to virtualize storage resources, maintaining a central repository that can be accessed across the network. We have designed a network block storage device, Peabody, that exposes virtual disks. These virtual disks provide mechanisms to: recover any previous state of their sectors and share backend storage to improve cache utilization and reduce the total amount of storage needed.

Peabody is exposed as an iSCSI target, and is mountable by any iSCSI compatible initiator. Using our implementation of Peabody, we show that for our workloads, up to 84% of disk sectors written, contain identical content to previously written sectors, motivating the need for content-based coalescing. *The overhead for writing in a simple implementation is only 20 percent of the total write speed.*

This paper describes our early experiences with the Peabody implementation. We quantify how rapidly storage is consumed, examine optimizations, such as content-based coalescing *and describe how recovery is currently implemented. We conclude with future plans based on these measurements.*

1. Introduction

Disk space on desktop computers has increased in size 1000 fold in the last ten years [7]. With the increasing capacity and reduced costs of storage, management has become one of the largest costs in storage and information systems outstripping hardware costs 3 to 1 [14]. Recovery and rollback provide ways to "undo" changes to a system. Much work has been done on the notion of "undo" at the file system layer, but few of these file systems are in widespread use. The problem is, writing file systems is complex and providing support for "undo" must be designed into the file system from the beginning. Logical Volume Managers (LVM) [21, 10, 6, 22] have included support for snapshotting, while the system is online, typically to support backup. The Network Appliance uFiler [15] provides periodic snapshots of its file system which is exported via NFS and CIFS. These two mechanisms are commonly used for high availability servers, but rarely applied to workstations because of high cost, high complexity (workstation users will typically not spend significant time designing their local storage architecture) or a lack of availability on certain platforms. Moving the "undo" mechanism into the *block layer* overcomes these problems by providing undo to any file system that implements a consistency checking mechanism, and utilizes network-based storage as its data store.

By virtualizing the disk, we can provide several benefits to any client operating system or application that can mount an iSCSI target. Virtual disks provide the illusion of a very large disk to the client. Peabody provides a centralized repository of sectors that allows for coalescing of replicated sectors across virtual disks to reduce space requirements. Finally in addition to storage virtualization and sector sharing, we also provide a mechanism for recovering any previous state of Peabody's exposed virtual disks. By maintaining previous versions of sectors and keeping a log of this state, we are able to roll the virtual disk state back to an arbitrary point in time, independent of the overlaying file system. The Peabody block storage system provides complete undo history and easy-to-manage virtualized storage for any file system or raw block device application (such as a database) that has a consistency checking mechanism.

The network-based storage device we have designed, *Peabody*, is implemented as an iSCSI target, and is mountable by any iSCSI compatible initiator. Peabody records a fine-grained log of each write transaction received. We implement semantic level "undo", (i.e. the ability to roll back an operation performed at a level above the disk, such as the file system), by noting that most file systems have been designed with mechanisms to recover when in an inconsistent state. For example, the FAT file system uses "`scandisk`" and Unix file systems use “`fsck`” to recover lost files and ensure meta data information is consistent. By maintaining a log of all writes to each virtual disk, we can create a mirror of the current state of any virtual disk, and then replay the log in reverse order to recover the state of the disk in the past. We then use the file system recovery mechanisms to make that mirror disk consistent. If the file system can not be made consistent at that point in the write log, the pre-consistency check mirror disk can be moved forward or backward in time and the consistency check reapplied. Rather than use the rolled back disk for data recovery, it can replace the current system disk, in effect, becoming the system disk at an earlier point in time.

We have successfully booted Linux from a Peabody disk and run Windows 2000 Professional Edition with Peabody as the system disk. (We ran the Windows system under the VMWare Professional Workstation virtualization environment since current iSCSI HBA's and client libraries do not allow us to boot Windows systems from an iSCSI target.) We have used Sherman, the Peabody virtual disk manager, to rollback changes on Linux systems using the `ext2`, `reiserfs`, `xfs` and `jfs` file systems. We have also recovered disks using the `NTFS` and `fat32` file systems for Windows installations.

Virtualizing disks and maintaining a long-lived write log incurs additional overhead. There may be properties of the way that drives are used that can be exploited to reduce those overheads. In this paper, we report on early experiments with our initial Peabody implementation. We find that, although Peabody reduces disk write bandwidth by up to 20%, application performance is virtually unaffected. We show that, for our workloads, up to 71% of disk sectors written to two virtual disks, contain identical content to previously written sectors, when two disks share the same Peabody server. Additionally, up to 84% of disk sectors written contain identical content to previously written sectors *within* a given virtual disk. This commonality can be exploited by physically storing a single copy of shared sectors. We also found that many writes are “silent writes”, writing sectors to disk, for which the content is the same as the existing sectors. Maintaining information about the *content* of individual sectors can also eliminate the need to perform these “silent writes”.

The remainder of this paper is organized in four sections. In Section 2 we examine the detailed experimental setup, our workloads, and our measurement methodology. We then discuss our results in Section 3. In Section 4 we detail related work and how it compares to our current research. Finally, we offer some conclusions in Section 5.

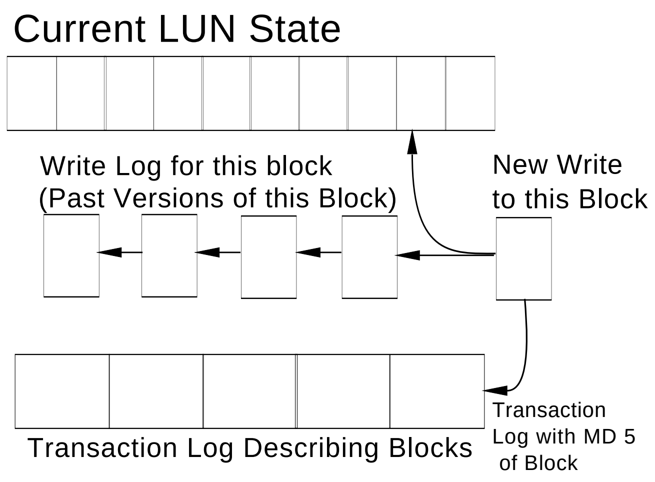

Figure 1. Peabody Write Log Design

2. Peabody Prototype: Design and Implementation

The goal of our first implementation of Peabody was to quantify how different file systems and operating systems use underlying storage systems. Other groups, such as the Storage Systems Program at HP Labs, have provided long traces describing the access patterns of disk drives used by different workgroups [18]. This data was collected by instrumenting the SCSI subsystem of workstations and recording a transaction log. In addition to type of access (read or write), and the sector offset, which Wilkes *et al* collected, we also need to know properties of the *content* of the data being stored.

We modified the Intel iSCSI reference implementation [12] to implement Peabody. The Intel reference implementation provides an iSCSI target as a user space process. Virtual disks are provided by exporting appropriately sized files or block devices as the “backing store” for a virtual disk. We modified that implementation to support 64-bit file system sizes up to 2 TB in size. We also modified the iSCSI target to record the contents of each write to a write log and to record various meta-data to a transaction log. This meta-data includes the location being written, time stamps and content summary hashes; See Figure 1.

In this paper, the content summary is used to determine if two sectors have identical content. We used the MD5 algorithm to compute the 128 bit content summary hash of each 512 byte sector. We chose MD5 because it is a universal hash with good hash distribution, and very low probability of collision; no hash collisions (*i.e.* the same hash for different content) were found in our traces. The OpenSSL [16] implementation of MD5 on an Intel Pen-

Table 1. Workloads used in evaluation of Peabody. Disk Sector size is 512 bytes. Workloads D-K use the `ext3` file system, while workloads L-Q use the `NTFS` file system.

Name	Workload	Disk Sector Write Count
A	XFS Application compile	1,400
B	EXT2 Application compile	1,560
C	VFAT Application compile	2,914
D	Mandrake 9.0 Application Install	347,874
E	Mandrake 9.0 Idling	1,130,580
F	Mandrake 9.0 System Install	4,647,918
G	Mandrake 9 kernel build	752,414
H	Red hat 8.0 Application Install	337,674
I	Red hat 8.0 Idling	330,272
J	Red hat 8.0 System Install	6,389,226
K	Red hat 8.0 kernel build	2,426,756
L	Windows 2000 #1 Application Install	1,155,216
M	Windows 2000 #1 System Install	1,540,234
N	Windows 2000 #2 Application Install	1,170,044
O	Windows 2000 #2 System Install	1,538,558
P	Windows 2000 #1 System Usage	260,146
Q	Windows 2000 #2 System Usage	128,644

tium 3 Coppermine Processor consumes 4021 cycles per MD5, and at 1 GHz can consume disk sector data at 127 MB/sec, or 261 thousand MD5s/sec. This scales with processor speed, because 512 byte disk sectors fit in cache. The Intel Pentium 4 Xeon processor consumes 5041 cycles per MD5, and at 2.4 GHz can consume disk sector data at 250 MB/sec or 511 thousand MD5s/sec. A 2.6 GHz P4 Xeon gets 270 MB/sec. (The MD5 library and/or the gcc compiler used to test the speed of computation of MD5s are likely still optimized for the P3 leading to the larger number of instructions on the P4.)

We evaluated a number of iSCSI initiators; however, all measurements shown in this document were taken using an unmodified Cisco iSCSI initiator, provided as part of the Linux kernel. The target system was a single-processor 866MHz IA-32 system with 256MB of RAM, 20GB disk. The initiator system was a dual-processor 866MHz IA-32 system with 256MB RAM and two 20GB local disks. The systems were interconnected using a Dell PowerConnect 3024 100Base-T ethernet switch.

2.1. Workload Descriptions

To test our hypothesis that content similarity exists at the disk sector level (the minimum write granularity to modern magnetic disks, 512 bytes in our case), we took several disk sector write traces of various operations, (see Table 1). We used iSCSI as our disk infrastructure because it gives us direct access to the sector level transactions, and has enjoyed widespread support among operating system developers and hardware vendors. For this paper, we took sector level traces of simple file system tasks under Linux, as well as full system installation and application-level tasks on two different Linux distributions and Windows 2000. All Linux installation and application level tasks were performed using the ext3 file system [11], while all Windows tasks were performed using the NTFS file system.

Using the Intel iSCSI target and the Cisco initiator, we were able to mount targets as standard SCSI disks under Mandrake Linux 8.2 (with a 2.4.19 Linux kernel). For the first three traces, we created a partition on the disk, formatted it with the XFS, EXT2 and then VFAT file systems. We then ran a script that compiles an application, makes a small modification and then recompiles the application, and similarly runs $\LaTeX$ twice for slightly different inputs. These are traces “A”, “B” and “C” and represent a simple micro-benchmark which we could use to evaluate sector level coalescing with the same workload but differing file systems.

For a more realistic test of sector level similarity, we then created a partition on the disk and used that partition as the first system disk when installing different operating systems. Since we were unable to directly boot a computer using iSCSI, we use the VMWare [23] server virtualization software when installing the operating systems. We were able to install both Red hat 8.0 and Mandrake 9.0 as well as two installations of Windows 2000 using this VMWare on Peabody iSCSI disk method.

We were able to record sector traces of the entire installation process for these three operating systems. These are traces “F”, “J”, “M” and “O” (See Table 1.) In addition, on the two Linux installations, we downloaded, compiled and

Table 2. Performance of Peabody Implementation *vs.* Intel iSCSI

Confi guration	Read Bandwidth	Write Bandwidth
TCP/IP (`netperf`)	11.8 MBytes/s	
Intel iSCSI	9.4 MB/s	6.7 MB/s
Peabody	7.6 MB/s	5.3 MB/s
Native Disk	29.6 MB/s	28.1 MB/s

installed a 2.4.19 Linux kernel from ftp.kernel.org. These are traces "G" and "K".

We also performed several application installations on the four systems. On the two Linux installations ('D' and 'H') we downloaded Netscape 7.0 and Yahoo Messenger and installed them. For the two Windows traces ('L' and 'N'), we installed Netscape, Yahoo Messenger, Microsoft Office XP, Frontpage 2002, and Winzip 8.1. We left the Linux systems to run overnight, collecting trace data for approximately 12 hours. The Linux cron jobs, which do security housekeeping and file indexing, ran during this time. No other applications were launched besides the X server. These are traces "E" and "I". Finally, we used the Windows installations as our desktops for one workday, and performed normal operations such as email, web browsing, instant messaging and document editing. These are traces "P" and "Q".

In addition to testing sector level similarity, we evaluated the basic performance characteristics of our prototype. We compared Peabody read and write performance *vs.* both a standard software-based iSCSI implementation, and the raw network throughput from the initiator to the target. Table 2 shows the peak network performance for a TCP connection between the two computers using the netperf network performance toolkit [5]. It also shows the full line network to disk bandwidth achieved when reading and writing 1 Gbyte of data to and from the raw iSCSI block device when using the unmodified Intel iSCSI and our Peabody implementation, as well as the bandwidth for the same operation to the raw IDE disk used as the backing store.

This prototype implementation is robust but has poor performance. The Peabody implementation has about 20% lower bandwidth for both read and write requests than the Intel iSCSI target. The Intel target commits all changes to disk prior to sending a write acknowledgement to the initiator; this synchronous activity maintains SCSI semantics but reduces performance. Peabody currently, must also write the contents to the write log, compute the content hash and write the transaction log. This increases the time until a write is acknowledged. Since the Intel iSCSI target is a user-space process, this configuration has the additional overhead of copying I/O requests between the kernel and the user-mode process.

Recovery is managed by a user interface, called *Sherman*, that allocates a new virtual disk and traverses the write log in reverse order, writing the old contents back to the virtual disk.

This initial implementation is designed to be flexible rather than high performance. This implementation has sufficient performance to provide a usable system for long-term studies of I/O activity.

3. Results and Analysis

The goals of our initial study are to determine *how* different file and operating systems use the underlying iSCSI storage and how those commonalities and differences can be exploited when implementing a more robust version of Peabody.

Our prototype implementation of rollback and recovery uses a single write-log to record write transactions and uses a single file equal in size to the virtual disk as the backing store. While this mechanism works well for the case of an iSCSI target, it is likely ill-suited for the additional complexity associated with Peabody. For example, because we keep old versions of sectors, and also would like to coalesce sectors that have identical content, using a flat file would add the additional overhead of copying data for each write. A tree data structure would be more efficient for writes, but slower for reads. Also, inter-LUN sharing will require a layer of indirection between the sector storage and the LUN/offset associations.

One goal of this paper is to answer the question of what data structures are appropriate when fully implementing Peabody. An important datum necessary to answer this question is how often the contents of different sectors and I/O requests are the same. It may be that many sectors of data contain the same contents, either within a disk or between disks. If this could be exploited, it may be possible to maintain a single copy of those sectors in any I/O cache on the iSCSI target or to suppress writing duplicate sectors by using references to existing sectors with the same contents. To quantify the maximum amount of disk space and sector writes we could eliminate in an implementation of Peabody which performs *content-based coalescing*, we use the metrics of "silent writes" (described below), and "sector coalescing". Sectors which are available to be coalesced are those which have been written at some previous time, during *this trace*, someplace on the disk. Using this definition, we cannot coalesce 100% of the sectors written over the course of an entire trace because at least one sector must be written with content not already on disk for that trace. (The first sector written in the trace is fresh, if all sectors write the same content.)

Figure 2 schematically shows three common scenarios that can be exploited by the Peabody storage server. The

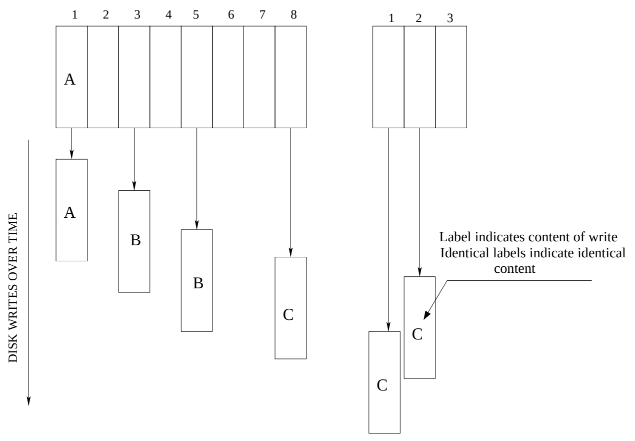
1
2
3
4
5
6
7
8
A
A
B
B
C
1
2
3
C
C
DISK WRITES OVER TIME
Label indicates content of write
Identical labels indicate identical content

Figure 2. Three common scenarios for coalescing sectors. The initial state of the virtual disks is displayed in the two arrays of rectangles along the top of the figure. The only content initially is content "A" in sector #1 of virtual disk #1.

two arrays of boxes across the top represents the initial state of the two virtual disks:

1. Sector #1 of the first disk initially contains the content labelled "A"; a later write to this sector also contains the same content. This is a "silent write" and does not need to actually be performed.

2. Two sectors with identical content "B" are written to sectors #3 and #5. These two sectors can be "coalesced" into a single write operation and an update to the "virtual disk" meta-data that indicates that sectors three and five are shared. This is an example of "intra-LUN coalescing", and is counted as one coalesced sector. Any sector with content "B" which is subsequently written to the first disk is counted as an "intra-LUN" coalesced sector as well.

3. Two sectors with identical content "C" are written to sector #8 of the first disk and sector #2 of the second disk. These two sectors can also be coalesced into a single write operation and an update to the virtual disk meta-data of both drives to indicate that those sectors are shared. This is an example of "inter-LUN coalescing", and is counted as one coalesced sector. Any sector with content "C" which is subsequently written to *either* disk (*e.g.* the write of "C" to sector #1 on disk two) is, by definition, counted as an "intra-LUN" coalesced sector because *that* disk has already seen *that* content.

How often do similar contents occur? (I.e. what percentage of writes are "silent writes" and what percentage of sectors are available for "sector coalescing"?) To answer this question, we used a micro-benchmark designed to see how different file systems affect sector content similarity. We also installed and ran several operating systems on Peabody to determine the affect of differing content and differing operating systems on sector content similarity.

3.1. File System Micro Benchmark

To determine the effectiveness of coalescing, we copied files onto the filesystems on the disk, ran a compilation with the Gnu C compiler ("gcc") two times and built a document using the $\LaTeX$ text processing system two times, each with slightly different input. These simple benchmarks produced repeated *content*, and we recorded whether this repeated content resulted in intra-LUN "sector coalescing", and/or intra-LUN "duplicate transactions". We define a "duplicate transaction" to mean that all of the sectors in the transaction were "coalesced sectors" *and* all of the sectors in the transaction had been written as a *single* transaction before. Note that this similarity is at the level of a *single* command to the disk, encompassing one or more contiguous sectors. We record "duplicate transaction" statistics for the file system benchmark to determine what granularity we will have to record content information at. Finally, this simple benchmark gives a hint as to whether or not repeated content will be sector aligned (and so available to be coalesced using fixed size 512 byte disk sectors as the quantum for coalescing) or not. We ran this benchmark for three different file systems `xfs` (Workload "A"), `ext2` (Workload "B"), and `vfat` (Workload "C").

Table 3 shows the results of this experiment.

For example, in workload "A", the `xfs` journaling file system caused 33 unique iSCSI write commands; of these, three transactions were "duplicate transactions". For workload "A", this results in 716,800 bytes of data being written in 33 transactions, of which 238,000 bytes were written in three "duplicate transactions". Because of file system implementation differences, the percentage of "duplicate transactions" varies between 3% and 30% for this workload.

The disk system actually stores data using 512 byte sectors on the disk. The percentage of sectors available for "sector coalescing" when measured in 512 byte sectors is very similar for each file system – about 97-98% for this workload, indicating that the common content is indeed being written in disk sector aligned chunks. However, the different file systems use distinct buffering and sector allocation strategies resulting in different sizes of iSCSI write command transactions. For example, the `VFAT` file system allocates files using 512 byte blocks (as hinted at by the large number of 512 byte iSCSI write commands for `VFAT` in Figure 3) while `ext2` uses 4096 byte blocks (as

Table 3. Intra-LUN Write Coalescing For Sample Workload – Coalesced operations are those that write content, which has already been written to the disk during that workload, either at the same offset, or another offset. Sectors are 512 bytes.

Workload	Total Transactions	Coalesced Transactions	Total Sectors	Coalesced Sectors
“A”	33	3	1,400	1,373
“B”	62	11	1,560	1,538
“C”	54	16	2,914	2,847

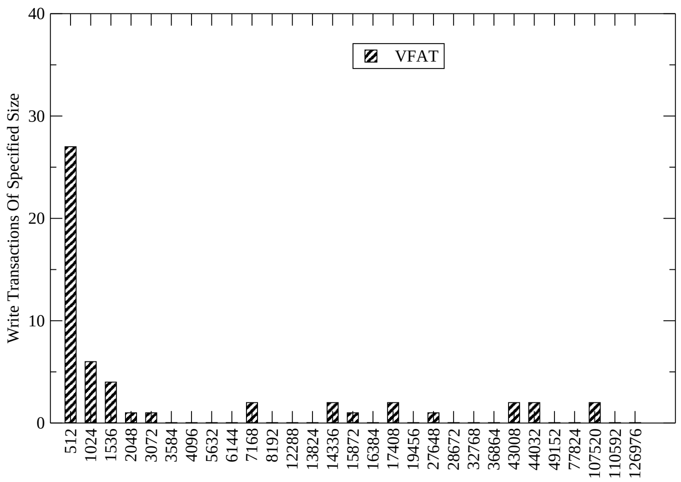

Figure 3. iSCSI write command size histogram for VFAT file system.

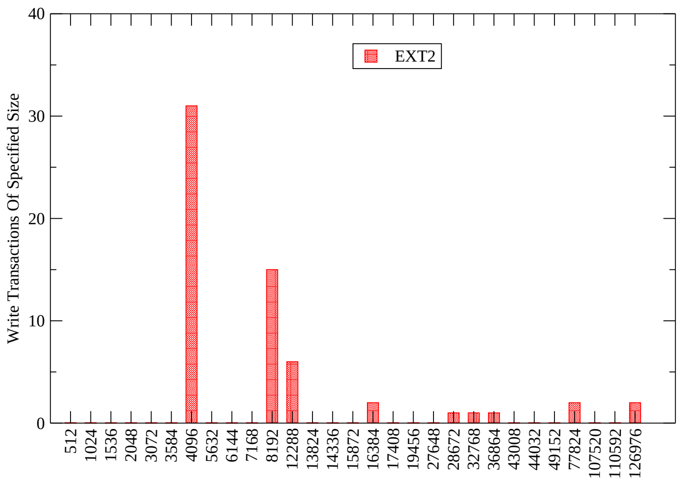

Figure 4. iSCSI write command size histogram for EXT2 file system.

indicated by the correspondingly large 4Kbyte iSCSI write commands in Figure 4). For this workload, the results indicate that maintaining fine-grained content summaries (i.e.

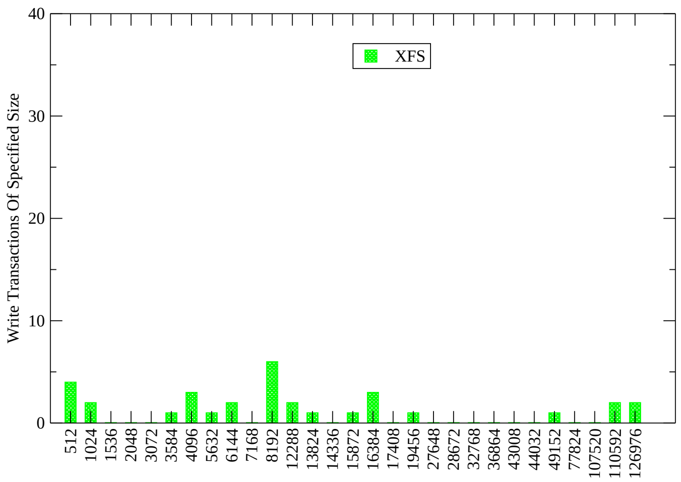

Figure 5. iSCSI write command size histogram for XFS file system.

for each 512 byte disk sector) will be necessary to achieve high content sharing.

This high degree of similarity is not surprising - in an earlier study, Ruemmler and Wilkes [18] found that disk writes only constituted ≈ 30% of disk requests and ≈ 60% of those writes were to file system meta data rather than file contents. Although our results are for different file systems, the results of Ruemmler and Wilkes may explain some of the duplicate writes. For example, our traces indicate that the `ext2` file system issues seven writes when creating a 1KByte file; one of these writes the actual file contents while the other writes update the directories and file system meta-data. The meta data is written in 4KByte blocks, although only a few bytes are modified. If we can efficiently represent these writes at the level of 512 byte sectors, this set of seven 4KByte blocks may increase the write log by seven 512 byte sectors rather than the 28KByte that would be indicated by the I/O transactions.

3.2. General System usage

The micro-benchmark tells us something about what we need to track to find common sectors (sectors available for “sector coalescing”) – content summaries should

Table 4. Total 512 byte sector writes for each workload, showing how much I/O Peabody is capable of eliminating. Coalesced sectors are writes for which, the content exists already on disk, and represent possible disk space savings. Silent writes are writes for which, not only does the content exist already on disk, but it exists at the same LUN/Offset for the write being committed, and represent writes which can be completely suppressed. Total eliminated I/O is the sum of silent writes and coalesced sectors.

Workload	Total Writes	Total Eliminated I/O	Silent Writes	Coalesced Sectors
D	347,874	26.8%	2.0%	24.8%
E	1,130,580	84.2%	1.3%	82.9%
F	4,647,918	30.7%	13.6%	17.1%
G	752,414	41.6%	5.2%	36.4%
H	337,674	24.6%	2.3%	22.3%
I	330,272	72.3%	2.5%	69.8%
J	9,143,056	26.6%	9.2%	17.4%
K	2,426,756	53.3%	2.3%	51%
L	1,155,216	22.6%	1.4%	21.2%
M	1,540,234	19.1%	9.8%	9.3%
N	1,170,044	22.1%	8.0%	14.1%
O	1,538,558	19.2%	9.7%	9.5%
P	260,146	18.3%	1.9%	16.3%
Q	128,644	28.4%	2.1%	26.3%

be recorded for *disk sectors* and not *transactions*. To determine how much commonality occurs in practice we installed several operating systems, and ran normal workloads through the system. The results for each of these workloads are in Table 4.

The number of disk sectors available to be coalesced is impressive for workloads 'D', 'E', 'G', 'I', and 'K', with a maximum of 84.2% sectors written already being on disk, for workload 'D' (Mandrake 9.0 idling). These are all Linux system usage traces. These results are encouraging because systems (at least Linux systems) appear to provide significant opportunities for write coalescing. Much of this coalescing arises because the Linux operating system usually performs 4KByte I/O operations.

While the two Windows installations('M' and 'O') and one windows usage trace ('P') had relatively low write coalescing, the minimum is still above 18%. This lower write coalescing occurs because the Windows filesystems appear to perform I/O operations using 512 byte sectors rather than the 4KByte sectors used in Linux. These results indicate that the technique of content-based coalescing may result in significant savings across operating systems.

A hint as to why there is such a difference in "sector coalescing" may be seen in Figure 6, which shows the distribution of iSCSI write sizes for trace 'D' (Mandrake 9.0 installation), and Figure 7 which shows the same distribution for trace 'N' (Windows 2000 Application Installation). A significant percentage of the data written in trace 'N' is the maximum iSCSI transaction size of 128 Kilobytes (631,190 of the 512 byte sectors written or 54%). Notice also that the vertical scales are different for the two graphs by a factor of four. There may be some consequence of writing large transactions which reduces instances of "sector coalescing". We will be investigating this phenomenon further in future work.

However, even the modest 18% minimum achieved by the lowest scoring workload still saves a significant amount of disk space over a long period of time. For example, the trace which resulted in the maximum rate of sectors written across all of our traces is trace J, which wrote 6,389,226 sectors. The minimum amount of coalescing observed for a single LUN workload was 18% for trace 'P'. If a user has a write stream that "looks like" trace 'J' in size and trace 'P' in the rate of coalescing, then that user will write 2.5 GB of unique data to a storage array. With these assumptions, and if this is an average daily write rate, a 1TB storage array (which can be constructed using five 320MB disks) would be filled after a year – in practice, we expect much less space will be used.

In their earlier study, Ruemmler and Wilkes found that 85% of sectors that are written will be over-written within an hour, and 25-38% of sectors written will be updated within a second. We have seen similar behavior in our initial tests, and this indicates that we may make a trade-off between the growth rate of the write transaction log and the granularity of available filesystem recovery points.

The Elephant File System [19] keeps *landmark* versions of files around permanently, and deletes intervening versions of files over time to recover space. We may be able to sustain recovery over longer periods by performing a similar coarsening of writes to disk. For example, if a file system always performs the same set of operations on a disk

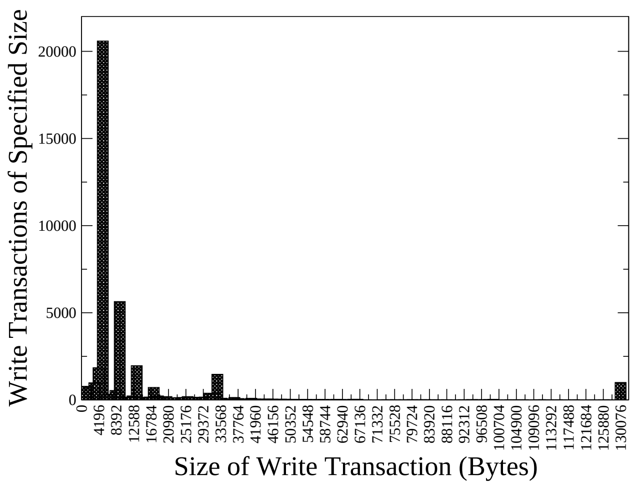

Figure 6. Trace D (Linux System Install)

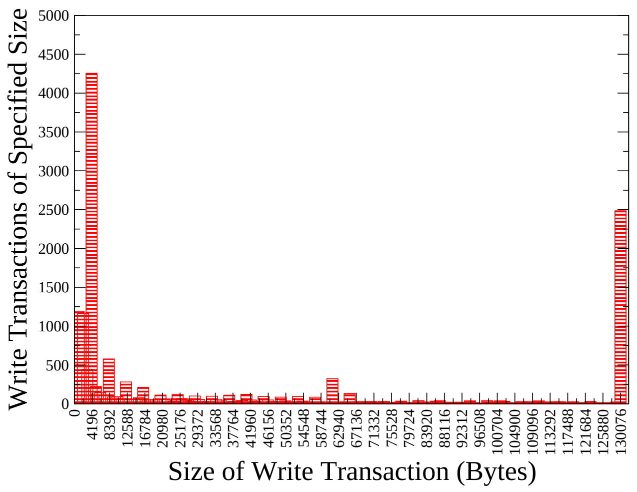

Figure 7. Trace N (Windows System Install)

when opening and closing a file system for write access, we can use these operations as cues as to when the file system is in a stable state and record this information as a *landmark* disk state. This information can then be confirmed using the normal filesystem verification tools.

On a more fine-grained level, we may be able to discard intervening writes to a sector, thereby recovering the storage associated with those sectors as well as increasing the recovery window for those sectors. By discarding intervening writes to a sector, recovery to the earliest time frame will still be supported, but the detailed log of changes to the disk will not be kept. Consider a Peabody system used over a period of a year. Normally *all* disk writes would be saved – the user could "roll back" the disk to any point in time during that year. However, the user may decide that they no longer need to be able to roll back to any specific millisecond in June. In this case, only the last write to any sector needs to be maintained between June and July. This "garbage collection" mechanism should save considerable space, but it needs to be integrated with the checks for file system consistency.

Finally, as an added benefit, all traces exhibited non-zero levels of silent writes, some above 10%. These are writes for which the content already exists on disk at the offset being written. An implementation which can track content, even a modest amount of tracking using an in-memory buffer cache, can eliminate a fraction of these silent writes, improving system performance.

3.3. Multiple Virtual Disks

After testing individual traces, for intra-LUN coalescing potential, we wanted to see if there was any inter-LUN sharing which occurred. We hypothesized that content similarity would exist for two distinct but identically executed Windows 2000 installations (Traces "M" and "O"), or even running similar workloads on two different workstations (Traces "G" and "K").

Sector categorizations can be: a fresh sector (not seen before in any LUN), an intra-LUN coalesced sector (seen before by the LUN with the reference), or an inter-LUN coalesced sector (not seen before by the LUN with the reference but seen before by another LUN). These are disjoint sets, so each sector is only counted in one category. Each time a sector is counted as an inter-LUN "coalesced sector" it is guaranteed to have different content than other inter-LUN sectors. Therefore, finding a significant amount of inter-LUN "coalesced sectors" indicates a significant amount of sharing of content, not just that there are lots of references to shared content between the virtual disks.

Another possible characterization would be to count a sector as an inter-LUN "coalesced sector" *each* time it is referenced, once there has been a reference in another LUN for that sector. This would give an indication of the number of references to shared data between virtual disks, but it does not give information about the number of actual distinct sectors shared between those disks. (Because a single sector which is shared between disks and then referenced repeatedly would show as a large number of references, but it is only a single shared disk sector.) Also, unfortunately, every time the sector is referenced after the first reference for that LUN, the sector is an intra-LUN "coalesced sector", and so the sector would be counted in more than one category using the above scheme.

An example clarifies these categorizations: a sector appears with content "1" in Trace "M", and then subsequently appears in Trace "O". The first reference to "1" in Trace "M" would be a fresh sector. Normally, the first reference to "1" in Trace "O" would also be a fresh sector, however when the traces are combined, we can find the sector "1" already on disk when it first appears in Trace "O", and this reference is recorded as an inter-LUN "coalesced sector".

Table 5. Total 512 byte sector writes for each combined workload. The "percentage of coalescing" column shows the fraction of sector writes that can be coalesced with earlier sector writes. The column labeled "InterLUN" are coalescing between LUN 0 and LUN 1. The columns labeled "LUN 0" and "LUN 1" are sector writes coalesced within a single LUN only. The column labeled "Total" is the sum of the interLUN, LUN 0 and LUN 1 columns. This table is sorted by the percentage of sectors common to both workloads (InterLUN percentage).

		Percentage of Coalescing			
Workload	Total Writes	Total	InterLUN	LUN 0	LUN 1
DH	685,548	58.8%	33.1%	13.6%	12.1%
MO	3,078,792	51.5%	32.3%	9.6%	9.6%
LN	2,325,260	54.3%	32.0%	11.2%	11.1%
GK	3,179,170	61.1%	10.6%	9.8%	40.7%
FG	5,400,332	35.9%	3.6%	26.4%	5.8%
FK	7,074,674	41.2%	2.7%	20.2%	18.3%
GJ	7,141,640	40.8%	2.3%	4.4%	34.1%
PQ	388,790	23.1%	1.5%	12.2%	9.4%
HI	667,946	49.1%	0.9%	12.4%	35.8%
IK	2,757,028	56.2%	0.7%	8.7%	46.9%
DE	1,478,454	71.2%	0.5%	6.3%	64.4%
DG	1,100,288	37.3%	0.4%	8.5%	28.4%
EG	1,882,994	67.6%	0.4%	50.6%	16.6%
LP	1,415,362	22.1%	0.3%	18.4%	3.4%
NP	1,430,190	21.7%	0.3%	18.1%	3.3%
NQ	1,298,688	23.0%	0.3%	19.9%	2.8%
LQ	1,283,860	23.4%	0.3%	20.3%	2.8%
EF	5,778,498	41.4%	0.3%	16.5%	24.7%
HK	2,764,430	50.0%	0.2%	3.0%	46.8%

All subsequent writes of content "1" in "M" or "O" are counted as intra-LUN "coalesced sectors".

The results for the several semantically similar pairs, as well as some semantically dissimilar pairs are given in Table 5. For example, workload "DH" has 685,548 sector writes, of which 13.6% can be eliminated by doing content coalescing on LUN 0, 12.1% can be eliminated by doing content coalescing on LUN 1, and 33.1% can be eliminated by doing content coalescing across LUN 0 and LUN 1. Therefore, 58.8%, (the sum of the three above percentages), of the sectors written during trace "DH" can be eliminated by doing inter-LUN content sharing and coalescing on all sectors.

Table 5 is sorted by the amount of inter-LUN sharing observed for the traces. The highest level of sharing occurred when we executed the same operation, on the same file system, on the same operating system. The top three examples are DH, which represents installing Netscape and Yahoo Messenger on `ext3` under Linux and shares 33.1% of its sectors between LUNs; MO, which represents the two Windows 2000 installations, which were performed as identically as possible, and exhibited 32.3% sharing between LUNs. Finally, trace 'LN', which represents the application installation traces under Windows, shared 32% between LUNs. These results are in line with previous work done at the file system layer with up to 550 machines [2].

This also makes intuitive sense, and provides a good argument for doing inter-LUN sharing when the operating systems and file systems used between LUNs are similar. Even when there is very little inter-LUN sharing, high levels of *intra-LUN* coalescing can still be achieved for the combined traces. An example is trace 'EG', which has only 0.4% inter-LUN sharing, but still has over 67% intra-LUN coalescing.

Finally, all pairwise executions produced common sectors. Some produced very few common sectors, as few as 4 sectors were similar when combining workloads I and M (Red Hat 8.0 idling and Windows 2000 installation).

3.4. Impact on Data Structures

There are four data structures needed to implement the recoverable storage in the Peabody storage server:

- The *buffer cache* provides in-memory caching of the contents of a virtualized disk.
- The *sector store* provides the actual storage for the virtualized disk as seen by the user of the storage server.

We assume each virtualized disk may either share a sector store common to all disks or use a private sector store.

- The *write log*, which is used to record the prior contents of sectors. Maintaining this write log allows storage volumes to be recovered – recovering involves "undoing" individual writes.
- The *transaction log*, which records the meta-data associated with the write log. While the write log records the *contents* of the writes, the transaction log records information such as the order of writes, when they occur, their location and content hashes of sectors written.

We will address each data structure in turn. For each structure, we need to determine if there would be any possibility of exploiting content sharing in that data structure and enumerate the design alternatives. In each case, we are interested in improving performance and decreasing the total amount of physical storage needed to represent a perceived amount of storage.

Since the Peabody storage server is designed to serve multiple virtual disks concurrently, it may be possible to exploit content similarity in the buffer cache. Buffer caches are usually 128MB-4GB for different storage servers; although this a small amount of memory, RAM can be accessed about a thousand times faster than disk. Thus, more efficient use of the buffer cache can greatly enhanced performance. We believe the data structures needed to exploit content similarity in the buffer cache are easy to maintain, and this represents a good opportunity for improving performance.

Improving the performance of the buffer cache may improve performance, but it does not reduce the total amount of physical storage necessary to represent a virtual disk. We feel there may be three possible organizations of the sector store. The first is to represent each virtual disk using a physically contiguous region of the actual available disk. This obviously involves no overhead, but also saves no space. It provides the best performance of the storage alternatives since the disk store can take advantage of spatial locality. This would be particularly true for workloads with many read operations. The traces we collected had a relatively low read/write ratio – this happens because the buffer cache on each computer filters out many of the reads.

Alternatively, each virtual disk may be represented as a sparse data structure, such as a B-tree of pointers to actual physical disk sectors. This organization would allow a great degree of sharing, but have the additional overhead of recording the sector structure. Furthermore, sectors that appear physically contiguous in a virtual disk would actually be distributed over the full disk; this may greatly decrease I/O performance.

The third alternative involves a middle-point between the other two designs. It may be possible to use a series of "extents", similar to the mechanism used in logical volume managers, to allow finer-grain control of storage sharing. Regions with high read usage or poor sharing could be represented as a physically contiguous extent while regions that have significant sharing can be represented using a sparse tree organization. If any mechanism to exploit content similarity is used, additional data structures will be needed to identify sectors with identical content.

We believe there is little benefit to sharing content in the transaction log because it is unlikely that the contents of the transaction log will have much similarity between virtual disks. However, there is likely to be considerable similarity within the write log itself. The write log can either be organized as a contiguous series of writes or as a sparse data structure. If both the write log *and* the sector store are organized as sparse data structure, it is possible to share content between the two structures; this would provide the greatest opportunity for reducing the physical storage needed.

In future work, we hope to implement multiple alternatives and compare the efficacy of each. Based on the analysis in this paper, we believe that maintaining the write log using a sparse structure is appropriate, because there is significant content similarity in the traces. It is not clear if implementing the sector store using a sparse mechanism will save enough space to justify the likely performance penalty. Our initial implementation will focus on the extent-based mechanism, since this is more likely to result in a compromise between space savings and performance.

4. Related Work

A few projects have attempted to address a subset of the features provided by Peabody. The Logical Disk (LD) [4] was an attempt to separate file system implementation from physical disk characteristics by providing a logical view of the block device. Jonge *et al* show how a LD can turn an otherwise unreliable file system into a log structured one. One of the performance optimizations they do is to compress blocks to better utilize disk space. Logical Disks were primarily proposed to improve disk performance, and do not discuss the notion of content-based views of blocks. Furthermore, block-layer rollback and recovery mechanisms were not explored, although [4] does propose multi-version filesystems as a topic of future work.

There have been various implementations of Logical Volume Management (LVM) [21, 10, 6, 22]. The common features include semantic separation of physical block devices from logical block devices allowing improved management of the logical block device over time, the ability to take a snapshot of a running system, typically using a copy-on-write mechanism, and replication of blocks (typically through mirroring) for fault tolerance. To our knowledge, no systems to date have the continuous roll-back mechanism of Peabody and snapshots in those other systems are typically read-only.

The Petal Project [9] at Compaq SRC provides a virtual block device interface implemented using many servers in a distributed fashion. Similar to LVM, the Petal virtual disks also provide an explicit snapshotting capability. It is accomplished by running a user-mode program that ensures the disk is in a consistent state before making the snapshot. They discuss the idea of a "crash-consistent" snapshot where a snapshot may be taken anytime while blocks are being written, and consistency is recovered using a file system consistency program, but they have reportedly not implemented the ability to do this. Obviously, snapshots are explicit, but they are also read-only. Petal is implemented on Digital UNIX as a virtualized disk service that appears as a standard disk driver to the operating system, making it an operating system specific system.

4.1. Versioning File Systems

File systems which maintain multiple versions of files have also been heavily studied. The VMS file system allowed for previous versions of files to be accessed via a hidden directory where the file was located. The Network Appliance uFiler [15] product allows snapshots to be viewed as read-only hidden files in an associated directory. Snapshots in uFiler are scheduled at regular intervals to perform similarly to the continuous mechanism of Peabody, however the granularity can be such that data is lost.

For example, if one snapshot interval is five minutes, the "every five minute" snapshot represents the state of the file system as it was recorded sometime within the last five minutes. If a file is written three times in less than any five minute span, then at least one of those writes will not have been recorded in a snapshot. This is true because the uFiler only takes a snapshot every five minutes. If write 1 occurs, and then the snapshot is taken, then writes 2 and 3 will occur before the next scheduled snapshot. If write 1 and 2 occur before the snapshot is taken, then write 3 will occur before the next scheduled snapshot, (and write 1 is not recorded in a snapshot). If all three writes occur before the snapshot is taken, then only write 3 will be reflected in the next snapshot, (and writes 1 and 2 are not recorded in a snapshot). Finally, snapshots are typically discarded when the next snapshot for that time interval is taken, so after ten minutes, the "every five minute" snapshot taken ten minutes ago is no longer available.

The Comprehensive Versioning File System (CVFS) [20] keeps old versions of files to allow for security rollbacks in case of intrusion. The Elephant File System [19] maintains versions of *files* and uses various policies to decide when to delete these files.

The problem with all of these file systems is just that; they are file systems. They must be ported to different operating systems to be useful and they provide only the file system semantics and abstractions provided by that file system.

We believe that in order to be widely useful the versioning mechanism needs to be in the block layer rather than file system layer. Peabody can provide the snapshot mechanism of the uFiler, the security rollback mechanism of CVFS, as well as the possibility of deletion of old blocks using semantics like the those of the Elephant file system.

4.2. Other Related Projects

Venti [17] describes a system that uses hashes of blocks to coalesce writes to the actual data store to save space. However, there are several important differences between the current implementation of Venti and Peabody.

Because Venti is used as a backup device, it only takes snapshots every 24 hours. This prevents using Venti to roll back the state of the disk to arbitrary points in time, as is possible in Peabody. However Venti could probably be modified to include this functionality. Storage is reduced in Venti by addressing each block using its MD5 hash. When a block is written, if it is already in the backing store, then it will simply be overwritten with the same contents. By definition, if the contents change, then the MD5 of the block will be different and the block will be stored in a different place. Finally, Venti is intimately tied to the Plan 9 Operating System, and so could not be used as the backing store for a Windows 2000 or Linux installation as we have demonstrated with Peabody, although again, Venti could probably be modified to provide this functionality as well.

LBFS [13] describes using "semantic block boundaries" to collapse chunks of files which are the same to reduce write bandwidth. Whenever a file is written, the server computes block boundaries not based on offset in the file, but rather by using a fingerprinting mechanism which has a reasonably small output space. (They use Rabin Fingerprinting [3] in the paper.) Each *byte* in the file is incrementally added to the potential block, and then the fingerprinting algorithm is run on that block. If the fingerprint output is some pre-chosen magical number, then that offset is output as the block boundary, and a new block is begun. The interesting property of this mechanism is that insertions or deletions in the middle of a file do not cause global changes to the blocks after that block in the file. Instead, there is a local change to that block, either producing two blocks, or combining a previous block with that block, but otherwise none of the other blocks in the file will change. This is important for the LBFS implementation because they are using the MD5 hash of the resultant "semantic block" to save bandwidth between client and server by not sending blocks which haven't been altered when a file is written out.

If semantic block chunking was implemented in Peabody, possibly much more content similarity could be discovered, but the overhead for computing the byte-wise block boundary calculation at file creation time might prove too much overhead to use. We will be investigating this in the future.

Many file systems attempt to put blocks that are related semantically, together on disk. It reduces disk arm seeks when reading, improving performance. We do not relocate blocks in the up-to-date version of the disk, to allow these optimizations to continue to produce performance improvements.

Berkeley's Recovery-Oriented Computing (ROC) relies on the ability to recover data [1]. They have implemented an email store that allows for undo to occur, but they use a conventional database to record the transaction log. Peabody seems like a natural block storage mechanism for use underneath any recoverable application.

Finally, OceanStore [8] is an attempt to provide secure, fault-tolerant, highly available storage in the network. Data is injected into the system using a new interface that is more full-featured and more high-level than the file system. They provide legacy support in the form of a UNIX file system wrapper on top of their higher level primitives.

5. Conclusions and Future Work

The results we obtained for the general purpose workloads indicate that there is a significant amount of content sharing within a single virtual disk – up to 84% at best. This is an important result because it validates our hypothesis that we can save space by using information about the content of the sectors to store identical sectors only once. There is also a significant amount of silent writing occurring.

Additionally, for the case where multiple virtual disks are sharing the same backend block storage, when the virtual disks are configured similarly, and operations performed on the disks have semantic similarity, there appears to be significant sector sharing – up to 32% for the best case.

These three results suggest an implementation which focuses on a unified sector store, as well as an in memory data structure which can detect silent writes and discard them.

While the workloads used represent a relatively small time frame, these results are encouraging enough that we will be implementing a more sophisticated version of Peabody which does sector coalescing and inter-LUN sector sharing to test our hypotheses over a longer period of time. It might be the case that there is some longer term self-similarity in the sectors written which Peabody will automatically take advantage of because of its sector history.

Additionally, we have come up with a *Content-Based Block Caching* scheme which utilizes the contents of sectors to store only a single copy of a sector in memory to improve cache read hit rate. We intend to marry the Content-Based Block Cache and Peabody to evaluate the performance of the combined system.

6. Acknowledgements

The authors would like to thank Jim Mankovich for his insights during the initial brainstorming around Peabody, as well as our shepherd Curtis Anderson for his helpful feedback and suggestions.

References

[1] U. C. Berkeley. Recovery orienting computing project. `http://roc.cs.berkeley.edu/`.

[2] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed fi le system deployed on an existing set of desktop pcs. In *SIGMETRICS 2000*, pages 34–43, Santa Clara, CA, June 2000.

[3] A. Z. Broder. Some applications of rabin's fi ngerprinting method. *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152, 1993.

[4] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: a new approach to improving fi le systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages pp. 15–28, Asheville, NC, December 1993.

[5] Hewlett_Packard. netperf home page. `http://www.netperf.org/`.

[6] IBM. Aix logical volume manager quick reference. `http://www-1.ibm.com/servers/aix/products/aixos/whitepapers/lvm\_ver.html`.

[7] K. Keeton and E. Anderson. A backup appliance composed of high-capacity disk drives. In *Hot Topics in Operating Systems VIII*, page 171, Elmau, Germany, May 2001.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[9] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[10] G. Lehey. The vinum volume manager for bsd. http://www.vinumvm.org/.

[11] Linux. ext2fs home page. `http://e2fsprogs.sourceforge.net/ext2.html`.

[12] M. Mesnier. Intel iscsi reference implementation. `http://sourceforge.net/projects/intel-iscsi/`—, Dec 2001.

[13] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[14] G. Nagle. The cost of owning and storing data. `http://www.storagesearch.com/costdata.html`.

[15] Network_Appliance. ufiler technology overview. `http://www.netapp.com/tech\_library/3001.html`.

[16] OpenSSL. Openssl home page. `http://www.openssl.org/`.

[17] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey, CA, 2002.

[18] C. Ruemmler and J. Wilkes. Unix disk access patterns. Technical Report HPL-92-152, HP Labs, Dec 1992.

[19] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[20] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in a comprehensive versioning file system. Technical Report CMU-CS-02-145, CMU SCS, May 2002.

[21] D. Teigland and H. M. S. S. Inc. Volume managers in linux. In *Proceedings of the 2001 USENIX Annual Technical Conference Freenix Track*, Boston, MA, June 2001.

[22] Veritas. Veritas foundation suite for hpux. `http://eval.veritas.com/downloads/pro/vxfs\_hpux\_whitepaper\_pdf.pdf`.

[23] VMWare. Vmware home page. `http://www.vmware.com/`.