THE UMBRELLA FILE SYSTEM:

STORAGE MANAGEMENT ACROSS HETEROGENEOUS DEVICES

A Dissertation

by

JOHN ALLEN GARRISON

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2010

Major Subject: Computer Engineering

THE UMBRELLA FILE SYSTEM:

STORAGE MANAGEMENT ACROSS HETEROGENEOUS DEVICES

A Dissertation

by

JOHN ALLEN GARRISON

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,   Narasimha Annapareddy
Committee Members,   Riccardo Bettati
                    Krishna Narayanan
                    Alexander Sprintson
Head of Department,   Costas Georghiades

May 2010

Major Subject: Computer Engineering

ABSTRACT

The Umbrella File System: Storage Management

Across Heterogeneous Devices. (May 2010)

John Allen Garrison, B.S., University of Tennessee, Knoxville

Chair of Advisory Committee: Dr. Narasimha Annapareddy

With the advent of Flash based solid state devices (SSDs), the differences in physical devices used to store data in computers are becoming more and more pronounced. Effectively mapping the differences in storage devices to the files, and applications using the devices, is the problem addressed in this dissertation.

This dissertation presents the Umbrella File System (UmbrellaFS), a layered file system designed to effectively map file and device level differences, while maintaining a single coherent directory structure for users. Particular files are directed to appropriate underlying file systems by intercepting system calls connecting the Virtual File System (VFS) to the underlying file systems. Files are evaluated by a policy module that can examine both filenames and file metadata to make decisions about final placement. Files are transparently directed to and moved between appropriate file systems based on their characteristics. A prototype of UmbrellaFS is implemented as a loadable kernel module in the 2.4 and 2.6 Linux kernels.

In addition to providing the ability to direct files to file systems, UmbrellaFS enables different decisions at other layers of the storage stack. In particular, alternate

page cache writeback methods are presented through the use of UmbrellaFS. A multiple queue strategy based on file sequentiality and a sorting strategy are presented as alternatives to standard Linux cache writeback protocols. These strategies are implemented in a 2.6 Linux kernel and show improvements in a variety of benchmarks and tests.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Reddy, as well as my committee members, Dr. Bettati, Dr. Narayanan, and Dr. Sprintson for their assistance and guidance throughout this research. Their aide was invaluable in accomplishing this research.

I would like to thank all of my friends who have made my time at Texas A&M University such a positive experience. I would particularly like to thank the friends I have met at the Wesley Foundation for their support and friendship while I have been in College Station. I would also like to thank Max Mertz for the friendship and guidance he has shown to me in my time here.

I would like to thank my parents for helping to instill in me the desire to learn and to always do my best. I would also like to thank them for their love and support throughout my education.

Finally I would like to thank my wife, Briana, for her love, support, patience, and dedication. Finishing this research would have been much more difficult without her assistance and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Novel storage devices based on flash memory are becoming available with price/performance characteristics different from traditional magnetic disks. Storage systems in the future will likely incorporate both flash-memory based devices and magnetic disks. The unique characteristics of these different devices give cause to reexamine and potentially redesign the various layers of the storage stack. While current storage systems exploit devices of different characteristics, some of the characteristics of flash drives such as limited write cycles warrant the revisiting of this problem.

Two major types of Flash memory are in production. They are referred to as "NOR" and "NAND" Flash based on the way the transistors in the Flash are arranged to either resemble NOR or NAND gates. Both use floating gate transistors where the charge on the gate determines a logical "1" or "0." Typically NOR Flash memory is used more as a substitute for main memory and is byte addressable, while NAND flash is page addressable and is typically used in block-interface devices. NAND flash is used in SSDs and all subsequent references to Flash memory in this dissertation are directed towards NAND flash.

Within NAND Flash there are two additional approaches to storing data. The first is a single level cell (SLC), where there is only one charge level on the floating gate.

_____

This dissertation follows the style of *ACM Transactions on Storage*.

A newer approach is the multi level cell (MLC), where the amount of charge on the gate is used to indicate more than a binary state. MLC Flash is obviously capable of much denser storage and is thus less expensive than SLC Flash. SLC Flash is typically faster than its MLC counterparts.

Modern Flash SSDs are designed to function as plug in alternatives to traditional magnetic devices with a block interface. Even though the underlying mechanics are different, compatibility concerns force Flash SSDs to emulate the interface of traditional block devices. However, there are some fundamental differences between SSDs and magnetic disks. In a Flash SSD, it is not possible to write multiple times to the same area without first "erasing" the data that is present. In general, the write interface is emulated by using the erase and write functions. These devices also have a limited number of times they can perform the write operation without degrading the storage medium such that that area of the device becomes unusable. Writes to SSDs typically employ a copy on write strategy, writing the new data in a new location on the drive. A "flash translation layer" (FTL) in the physical device provides the conversion from block commands to their implementation on the Flash device. This layer also provides the mapping translation from the block level to where data is actually stored on the Flash drive. Some remapping is always involved in FTLs, so all actions, including reads, need to use the layer to arrive at the correct location. There are a variety of FTL policies involving buffers, specified log blocks, and other strategies designed to improve performance.

Table I: Operational latency of flash SSDs

Operation	Access Time
read	~25 µs (2 KB)
write	~200 µs (2 KB)
erase	~1.5 ms (128 KB)

The various underlying flash operations, in particular read, write, and erase, take drastically different amounts of time to complete. Example operation times from a Samsung flash device are included in Table 1 [Lee 2009]. These operations also operate on different amounts of data. Reads and writes take place in units of a page, which varies in size depending on the device. Erases operate on groups of pages called blocks. The erase operation which must precede an over write of data takes orders of magnitude more time than the write operation itself. Read operations are typically faster than write operations. In a traditional magnetic disk, sequential read performance is drastically better than random read performance. Because of the lack of moving parts that causes such a penalty to magnetic disks, Flash SSDs performance for random reads is much closer to sequential reads. Because of the order of magnitude latency difference between the underlying erase and write mechanics, as well as the limited number of erase cycles, FTLs are designed to minimize erases. If a drive is forced to erase a block before it can write a new block, its write performance can suffer greatly compared to either its "normal" operation when it is not waiting on an erase or when compared to magnetic

devices. Flash drives also typically have more uniform performance with respect to write size, where magnetic disks typically perform better with larger write sizes, as can be seen in Chapter V.

Flash SSDs in general have been shown to have variable write performance with respect to sequentiality. Random write performance tends to be quite poor, and when a workload mixes random writes with reads, overall performance can suffer quite significantly. Additionally, some Flash SSDs have a block crossing delay that is incurred when subsequent writes cross certain block boundaries in either direction [Chen 2009]. This penalty is similar to a seek penalty in magnetic disks, although clearly not caused by the arm moving to the appropriate location as in magnetic disks.

As an example of the boundary crossing penalty, Figure 1 shows a stride test on a Transcend TS16GSSD25-S 16GB drive, although results are similar for the Memoright and Samsung drives used in later tests (data from [Dunn 2009]). Blocks of 4KB of data were written at 32 KB offsets throughout the drive and the latency of each operation was collected. Latency spikes are observed at 1 MB offset intervals when using various stride sizes, leading to the conclusion that some of the SSD drives suffer a boundary crossing penalty. Even when these boundary crossing penalties are not present, the writes require more attention in SSDs because of the reasons mentioned above.
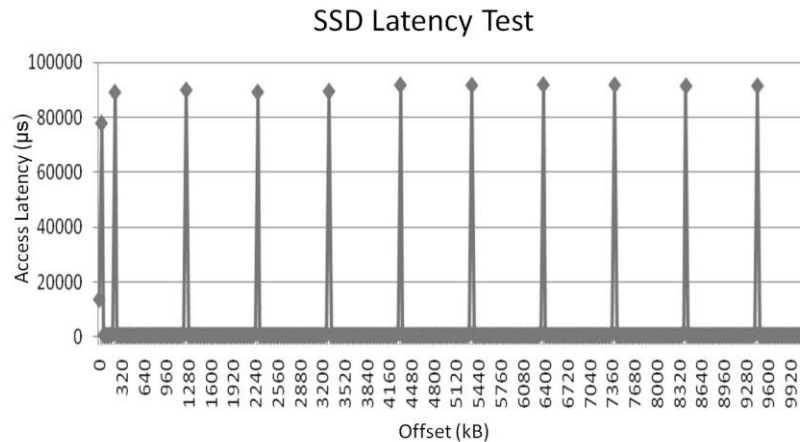
Fig. 1. Boundary crossing penalty in SSDs.

Like devices, applications also have remarkable diversity. This diversity impacts performance and provides an opportunity for improvement of the storage system. While some applications utilize primarily sequential storage accesses and thus would not benefit from caching, some other applications have much more locality in their access patterns and would benefit greatly from caching. Some files may be read-only while others may be frequently updated.

Current file systems map files to devices based on the namespace organizations. Typically, a file system's data is mapped to a fixed set of devices based on the way the namespace is mapped to the underlying devices. If a user wants to locate some of his files (say /usr/foo) on one device (a flash drive) while locating other files in /usr on another device, most current file systems do not provide a simple mechanism to allow this since all of the files under /usr are mapped to a fixed set of devices in the storage

system. This mapping is determined by the relation between the namespace and devices in the storage system.

In general, current file systems do not provide applications above them a choice of where to locate files or which devices to employ. This choice is only available at the entire file system granularity and not at an individual file level. Work presented in Chapter III proposes a solution to allow diversity of applications to be matched to the diversity of devices at an individual file level. The matching of application characteristics to device characteristics can be done by a system wide policy or based on the user's preferences.

As an example configuration that can benefit from matching device characteristics to the data and applications that use them, consider a system with three different types of storage, traditional magnetic storage devices, a smaller capacity flash based device, and another magnetic disk with hardware encryption support. There are various file systems [Halcrow 2005; Hohmann 2007; Gough 2006] and even device level [Seagate 2007] encryption options available to users. Consider a situation where some user files need to be encrypted on such a system. If an encryption file system encompassing all the devices is employed, the hardware encryption support available at one of the devices is not utilized to the extent possible (or some files may be encrypted twice). Encrypting all the files may impose overheads that may be undesirable for multimedia files and other files that do not have major security requirements. If normal file systems are employed on all the drives and device-level encryption is relied upon, the user has to carefully place the sensitive files on this device to ensure their protection.

Directing files to the appropriate underlying storage device while allowing the files to be viewed in the same directory would be a boon for this system. This would avoid the overhead of encrypting everything, as well as allowing the user to map files that best utilize the encrypting device for storing sensitive documents and the flash drive for storing read-only files (for example). Keeping all of this in a unified directory structure can ease the user's burden of maintaining the files on separate drives.

It has been observed that current device level coding schemes utilized to protect bit errors on the drive surface may not guarantee 100% data protection as drive capacities get larger. While data files require 100% bit accuracy, multimedia files can tolerate some bit errors without suffering any significant loss of quality. It is possible that in the future drives with different levels of bit error protection may become available. Again, mapping this diversity in devices to the applications using the data could provide numerous benefits to the user.

Additionally, benefits can be obtained by applying different policies at other levels of the storage stack than just the file system. With respect to Flash SSDs, there are major differences in write performance and behavior compared to magnetic disks, the traditional end point of the storage stack. A representation of the storage stack is shown in Figure 2. An application's write would traverse the various layers until ultimately arriving at the disk. Chapter IV focuses on write destaging to the device and how policies might be redesigned to best accommodate the basic differences at the end point of the storage stack. The differences in write and read performance, the relative importance of write performance, and the introduction of possible boundary crossing

penalties warrant revisiting the page cache management algorithms for writes when the underlying file system data may reside on an SSD. In particular, the work from Chapter III is utilized to differentiate which pages should be treated by a particular page cache writeout algorithm. Despite the particular application, the practice could be extrapolated to other areas of the storage stack.
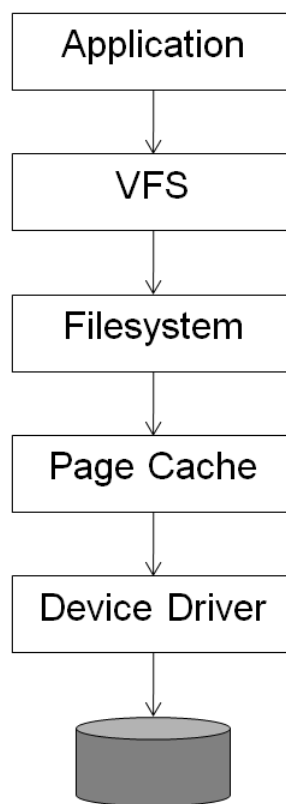
Fig. 2. Storage stack.

Much attention has been paid over the years to managing pages in a page cache. The focus of much of this work is in reducing the number of accesses of the storage device because of significant performance differences between DRAM and magnetic disk drives. Since applications typically issue synchronous read requests (where applications stall on the reads to complete) and the writes are completed asynchronously from the page cache to the storage device, the page cache management tends to focus on reads. Relatively little attention has been paid to which pages are being written out. This is because dirty page eviction is seen to have relatively little effect on overall application performance since these writes are completed asynchronously to the application. However, as memory access and CPU core speeds have increased, disk accesses have increasingly become the bottleneck [Jiang 2005]. While this has traditionally been thought of as a read throughput issue, in write heavy or mixed workloads, writes can quickly become a part of the problem.

With respect to the contents of the page cache, there are a few competing principles regarding whether or not to write pages to disk. Ideally, pages would be held in the cache as long as possible so that pages could be grouped together and sent to the disk in an optimal order. The more pages present that can be reordered, the better this can be done. Also, if pages are in the cache longer, it is possible that the data might be overwritten, thus saving a write to the disk. On the other hand, if the cache is devoted more toward write caching, the read performance would suffer. Additionally, when a program writes to disk, it typically considers that data to be "safe." If data were held for a very long time in the cache, the user might think that their data had been saved

permanently, but a power outage could still wipe out their data because it has not yet made it to nonvolatile storage. Clearly there needs to be some balance between caching a large amount of write data to improve performance, still having enough room to cache read requests, and sending data to the storage device at appropriate intervals so that the data is on a nonvolatile medium in a reasonable amount of time.

These policies have been developed with the concept of the data ultimately residing on a magnetic disk. Because of the significant differences in writes to Flash SSDs and magnetic disks, if Flash SSDs are to be the end point of the storage stack, additional effort should be directed to development and implementation of appropriate policies to reduce the impact of writes on SSD performance. In addition, reducing total writes to SSDs has important ramifications in improving the lifetime of the device.

Opportunities like this and the others a user could conceive serve as motivation for UmbrellaFS. With UmbrellaFS, users can select file systems that serve the particular characteristics of the data sets without compromising the user's convenience of organizing files any way he or she sees fit. Different files can be stored in separate file systems while being presented to the user in a single directory. The user can deal with a variety of files without worrying about the particular file system where a file is stored. With the UmbrellaFS rules framework, users have the option of setting up the mapping from files to underlying storage in a manner that optimizes file usage.

Chapter II presents related work. Chapter III presents a new layered file system that is designed to address device level differences, UmbrellaFS. Chapter IV presents modifications to the page cache writeback strategies. Results of benchmark tests of

UmbrellaFS and the page cache modifications are presented in Chapter V, and Chapter

VI concludes.

CHAPTER II

RELATED WORK

Device Differences

The considerable difference between Flash based devices and magnetic disks directs the efforts of this research to explore methods to take advantage of these differences. Various methods have been proposed to compensate for and exploit diversity in device characteristics. A wide variety of these methods have been proposed to affect the device or device driver level.

Device Level Approaches

Methods that work at the device level present a single view of the device to the file system and do not expose the diversity of the device characteristics to the file systems and applications above the file system. Two traditional approaches to device differences are to migrate data between devices with different characteristics and to use one device with different characteristics as a cache for another device.

HP's AutoRAID system is a device level approach to managing storage across RAID 1 and RAID 5 devices, and it is an example of data migration in response to device differences. In AutoRAID "hot" and "cold" data are stored in the RAID 1 and RAID 5 arrays, respectively. Blocks of data are automatically migrated between the two arrays as the characteristics of the blocks changes.

Recent work [Wu 2009] has targeted migration between Flash and traditional magnetic devices. Multiple devices are presented to the higher levels as a single device,

and a device driver migrates data between the devices based on a cost function and measurements over time. This way the blocks of data that benefit from the characteristics of a Flash device are stored on a Flash device, and those that benefit from being stored on a magnetic disk are stored there.

Intel's Robson technology [Trainor 2006] has proposed using NAND Flash RAM in order to operate as a cache for traditional hard drive accesses. By storing files in Flash RAM, the amount of time the system spends rotating disk arms on traditional hard drives can be dramatically reduced.

Flash storage in particular has been receiving increased attention. Policies for wear leveling, block level management and uniformity improvement have been studied [Baek et al. 2007; Gal and Toledo 2005; Kim and Ahn 2008; Kim and Lee 2002; Lee et al. 2007; Chang 2007].

File System Approaches

In addition to lower level approaches to device diversity, a number of efforts have been directed toward file system approaches. A number of these propose new file systems, and some others propose a layered approach similar to that taken by UmbrellaFS.

Sun's ZFS file system [Sun Microsystems 2007] allocates blocks flexibly across a pool of storage devices at the time of writing the file, rather than tying the file system to a device at the time of creating the file system. However, ZFS doesn't provide the user the flexibility of allocating a file across available devices.

Conquest File System [Wang et al. 2006] is designed to span both persistent memory and magnetic disks. UmbrellaFS can provide similar functionality in terms of the placement of files based on size, but cannot easily place metadata on one drive and data on another due to metadata consistency and underlying file system issues. UmbrellaFS has numerous additional rule possibilities in addition to Conquest's file-size threshold approach.

Panasas's storage allocation policy scales the striping granularity based on the file size [Panasas 2005]. Panasas does not have the same breadth of policy choices that are possible in a layered file system. The Veritas File System [Symantec 2007] has policies which can direct files to particular underlying storage devices. Unlike UmbrellaFS, the enforcement of these policies to move files between devices is not done on the fly as situations change, but must be directly triggered by an administrator. IBM's General Parallel File System (GPFS) is a high-performance parallel storage solution. GPFS provides users flexibility in striping policies for individual files. UmbrellaFS can provide the types of benefits from these file systems without necessitating a total switch to a proprietary file system. UmbrellaFS can direct files to appropriate file systems mapped to the underlying device characteristics.

Duke's Active Names [Vahdat et al. 1999] framework is another system that can be leveraged in a way similar to UmbrellaFS. In this case, however, to be used as a file system, the Active Names system would need to operate as its own file system. In addition, the Active Names framework focuses exclusively on the namespace.

UmbrellaFS can make decisions based on both the namespace and other metadata such as file modification times, permissions, and the like.

Unionfs [Wright et al. 2004] provides a similar unification of underlying file systems that UmbrellaFS does. Unionfs does not have the ability to target particular files to particular underlying devices, it merely combines existing file systems and presents a unified directory structure. An expansion on Unionfs, RAIF [Joukov et al. 2007] enables policy driven striping of files across file systems. In effect, RAIF uses different underlying file systems like a RAID array uses devices. It allows a policy driven direction of files to the different types of striping across file systems, but RAIF does not use file metadata in making these decisions. Decisions are based only on filenames, and policies are targeted to arrays of file systems, not individual file systems.

Other Methods

Brick based storage systems such as Self-* [Ganger et al. 2003] and FAB [Frølund et al. 2004] typically distribute file blocks randomly or based on a policy across the available storage bricks or nodes. However, this policy is typically not tailored to nature of the file or other characteristics of the file. The Google File System [Ghemawat et. Al 2003] randomly distributes the blocks of a file across a number of nodes.

While Object Storage Devices (OSDs) [Project T10 2004] can potentially provide such flexibility, OSDs require extensive changes to storage systems, file systems and clients. High level hints have been shown to be valuable in improving I/O performance [Patterson et al. 1992].

Policies have been employed in managing grid resources [Bharathi et al. 2005] and in enforcing security [Blaich et al. 2007].

OSD framework potentially allows storage systems to be tailored to specific characteristics of objects. However, the current OSD architecture or framework leaves the "policy" of allocation open and we are not aware of any specific studies investigating the utility of different allocation policies in OSD systems.

Packet interposing [Anderson et al. 2000] has been shown to provide benefits in scaling performance in a clustered file server.

Page Cache

Many algorithms have been proposed for managing page caches or storage caches, for example [Chen et al. 2005; Forney et al. 2002; Gill and Modha 2005]. These efforts have largely focused on improving cache hit rates. While both frequency information (LRU) and recency information (LRFU) [Lee et al. 2001] have been used in cache strategies, others have attempted to incorporate both recency and frequency. LRU-K [O'Neil et al. 1993], 2Q [Johnson and Shasha 1994], ARC [Megiddo and Modha 2003] and others have attempted to balance between the recency and frequency components of page behavior with various levels of overhead and adaptability. DULO [Jiang et al. 2005] examines temporal and spatial locality in making cache decisions. LIRS [Jiang and Zhang 2002] examines inter-reference recency instead of simply examining recency as the more commonly used LRU does.

All of these efforts have targeted cache performance with a focus on what to keep in the cache and what to discard. DULO in particular uses two separate sections to

try to increase sequentiality of accesses going to the disk. In their work and their implementation, however, they focus on the pages that are clean. In Linux, the order of pages on the various inactive and active queues does not affect the writes the system performs to the storage device. The work in Chapter IV is similar in that it uses multiple queues and sorting of those queues, however the focus here is on write order instead of reads, and this makes the work presented here complementary to these other efforts.

Efforts have been made to improve the power consumption of disks through different caching and prefetching policies. Papathanasiou and Scott propose to increase burstiness to provide more power down opportunities for hard disks [Papathanasiou and Scott 2004]. Zhu et al. propose both an offline power-aware algorithm to provide improved lower bounds on energy consumption and an online power-aware algorithm based on the insights from their offline algorithm to save disk energy [Zhu et al. 2004]. PB-LRU proposes a strategy to improve power consumption in multiple workloads with minimal parameter tuning [Zhu et al. 2004b]. These strategies are focused on traditional magnetic disks.

Efforts have also been directed toward cache writes, and these efforts bear the most resemblance to the work presented in Chapter IV. Some of these efforts have been directed toward buffers at disks themselves, and some are directed more at the caches within the operating system, but these approaches are all in general directed at controlling writes to improve performance.

STOW [Gill et al. 2009] separates writes into sequential and random queues and adaptively sizes the two queues based on workloads in a write cache. While this work

also divides workloads into sequential and random, it additionally sorts the queues and focuses on the page cache as opposed to a separate write cache in front of a storage array. Additionally, by implementing policies in the page cache, it is possible to gain access to additional information for decisions that is not available in the write caches for STOW. In a non-volatile write-specific cache considered in STOW, synchronous writes occur when the cache is full. In a general page cache, dirty page eviction is not so flexible and is controlled by the need to move dirty pages to stable storage in a reasonable amount of time. This leads to a need for solutions not only in the write caches of storage controllers closer to the disk, but higher in the page cache as studied here. Also, addressing writes at the page cache level can reduce the number of synchronous writes an application has to do, which is not possible at the storage controller level.

AWOL [Batsakis et al. 2008] improves page cache performance with respect to writes through the use of ghost caching and combining the memory manager and the I/O scheduler. This work has a similar goal of improving locality, although the separation of the page cache and the I/O scheduler is maintained. In addition, this research considers SSDs. This work differentiates between types of writes in that files written sequentially are treated differently than those written in nonsequential patterns. AWOL adapts the high-low thresholds of dirty pages to control when the pages are offloaded to the storage device and to control the amount of memory allocated to write caching. The approach here has not modified the high-low thresholds and can be seen as somewhat

complementary to AWOL. Incorporating the ideas in AWOL into this work is left for future work.

WOLF [Wang and Hu 2002] examines the write buffer of log-structured file systems and attempts to improve the write order going to log-structured file systems in order to increase performance. While WOLF proposes a multiple queue system similar to the proposal in this dissertation of multiple queues in the page cache, their focus is solely on log-structured file systems. This research targets caches in general, although it could also be applied to log-structured file systems.

Log-structured file systems have been suggested as a possible solution for reducing the random write I/O problem. They are particularly well suited to SSDs with very efficient random read performance and poor random write performance. Log-structured file systems may also benefit from changes such as the proposed sorting method and separation of the writes into separate queues [Wang and Hu 2002].

CHAPTER III

UMBRELLAFS

UmbrellaFS functions as a stackable file system located between the Virtual File System (VFS) [Kleiman 1986] and the file systems that hold data on the underlying devices. File systems interact with both applications and the underlying storage systems, and their location at the intersection of these two systems with diverse characteristics makes them uniquely suited to take advantage of both device and application characteristics. While the file system does not have large amounts of information about the underlying device where it resides, the system's administrator typically is aware of device characteristics and differences. UmbrellaFS allows the device characteristics to be exposed at the granularity of a file system. Hence, matching application characteristics to device characteristics translates into placing files of different characteristics onto appropriate file systems. UmbrellaFS provides a mechanism for files in a single user directory to be located appropriately on multiple file systems or devices through a user directed policy.

UmbrellaFS can act to enable appropriate use of different underlying devices such as Flash SSDs, traditional magnetic drives, networked data storage, and others. Additionally, UmbrellaFS provides the opportunity to selectively apply policies at other layers in the storage stack for additional exploitation of the differences in physical devices.

Hourglass Model

The Virtual File System (VFS) [Kleiman 1986] in relation to user's namespace and underlying file systems and devices is shown in Figure 3. The VFS maintains a one to one mapping between the namespace and the underlying file systems. The VFS layer is at the waist of an hour-glass with potentially a large number of user directories above and potentially multiple file systems at the bottom. If one is in the directory "/user1", the files "/user1/file1" and "/user1/file2" are not located on separate file systems. This can be seen in Figure 3. This one to one mapping between the directories in the namespace and underlying file systems ties the user's perception of the system to the underlying device characteristics. If this one to one mapping could be broken, such that one directory in the namespace could contain files from many underlying file systems, then this could enable exposure of device characteristics to individual files and allow new opportunities and options for the user.

By inserting another layer into the kernel, below VFS and above the underlying file systems, it is possible to place files from multiple file systems in the same directory. This layer would look and behave like the underlying file systems to VFS, and look and behave like VFS to the underlying file systems. When VFS receives a command, it would pass it into this new layer, which would then redirect the command to the appropriate underlying file system. This method would allow both VFS and the underlying file system to continue to operate normally, and would not require any modification of those parts of the kernel. While VFS continues to provide the important function of demultiplexing among the many different underlying file system

implementations, the underlying layer, UmbrellaFS, provides the function of unifying the namespaces and demultiplexing the namespace operations among the different underlying file systems.
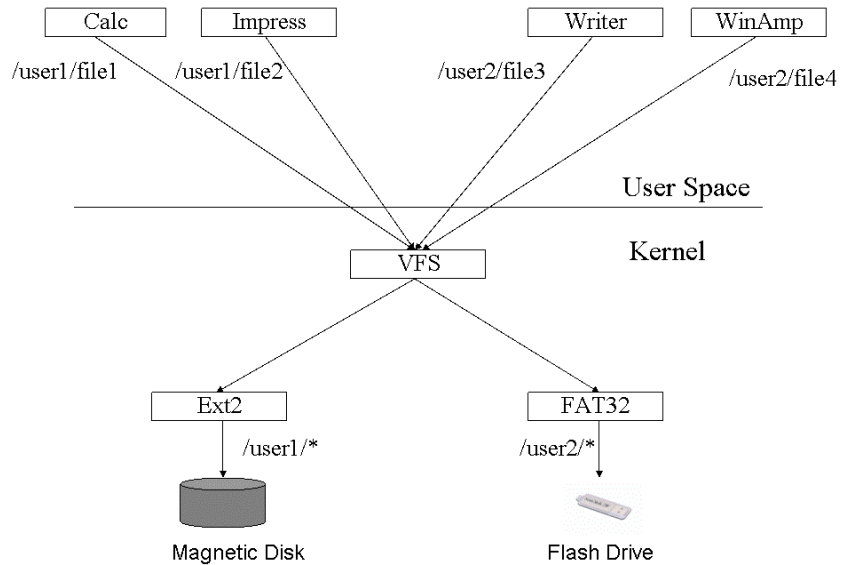
Fig. 3. Hourglass model with traditional virtual file system.

UmbrellaFS lies below VFS and interfaces between VFS and the underlying file systems. UmbrellaFS provides a unified name space for all the underlying file systems while also allowing policy based placement of files onto those file systems. The proposed approach with UmbrellaFS is shown in Figure 4. While VFS provides an hourglass model for file system implementations with a single interface, UmbrellaFS

provides a similar function for the namespace and individual users' files, as seen in Figure 4.
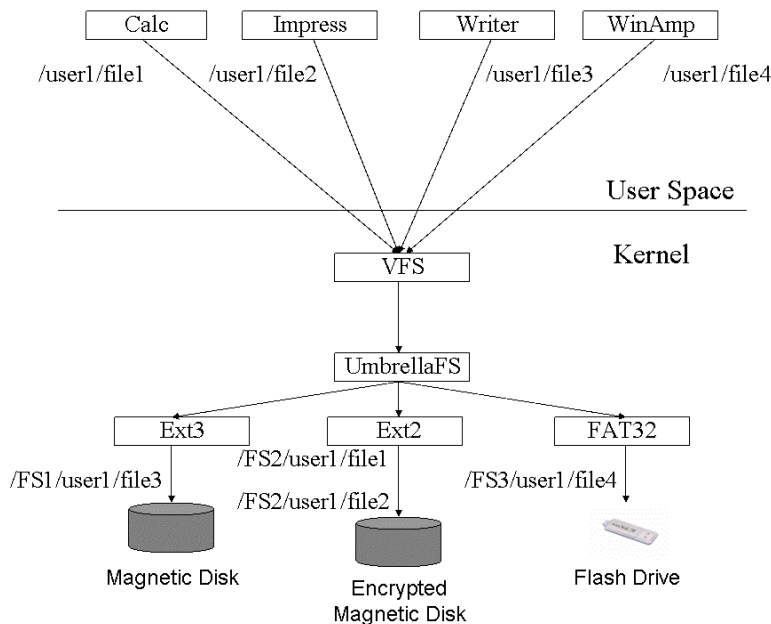
Fig. 4. Hourglass model with UmbrellaFS.

The Virtual File System provides a common API for applications to interact with file systems. This allows the file system's actual implementation of various functions such as writing, reading, etc. to be hidden from the application. For example, when an application accesses file /user1/file1 in Figure 3, VFS receives the system call and then calls the appropriate function for the underlying file system. When the underlying function returns, VFS returns that value up to the application. With UmbrellaFS, while the underlying mechanisms are slightly different, from the user's perspective the file

access proceeds as normal. UmbrellaFS appears as a single file system to VFS, and consequently to the user. When the user accesses the file, VFS calls the appropriate UmbrellaFS call. UmbrellaFS then translates the filename to the correct underlying file system and calls the appropriate function for that file system. When the file system function returns, UmbrellaFS returns the value to VFS as if it were the base file system on which the file resides, and VFS returns to the application.

In this manner, UmbrellaFS can leverage the disparity in device characteristics by mapping particular files to particular underlying file systems which appropriately exploit the device characteristics on which they reside. Additionally, UmbrellaFS increases the opportunity for future file system development with respect to particular underlying devices. Rather than requiring a full change of file systems, incremental development and installation of underlying file systems is permitted. UmbrellaFS exposes the device characteristics to the user or system administrator at the level of native file systems. For example, a traditional file system might be employed on a magnetic disk drive, while a log based file system is employed on a flash drive in order to conserve energy and lengthen the drive lifetime [Mathur et al. 2006]. Based on user-controlled policies, files can be mapped to the underlying device by mapping individual files (/user1/file1, /user1/file2… in Figure 4) to the native file systems (/FS1, /FS2… in Figure 4).

As an example of UmbrellaFS's utility, consider content specific storage. If the types of files are known, they can be mapped to an underlying storage device that best meets their typical access patterns. For example, UmbrellaFS can direct files to

underlying file systems based on the file suffix. Multimedia files with extensions such as .jpg, .gif, and .bmp files could be stored in /images. Text files such as .txt and .doc extensions could be stored in /text, and video files such as .avi files could be stored in /video. The user is presented with a single directory structure with all of the files present. For example, /user/dir1/foo.txt and /user/dir1/foo.jpg would be stored in /text/dir1/foo.txt and /images/dir1/foo.jpg respectively. With files mapped in this manner, it would be possible to put text files on a file system with redundancy, while the various multimedia files could be stored on a system with wide striping for faster access, etc. Figure 5 shows the user's view of the namespace, as well as how the files are stored in the underlying storage.

There are alternatives to the implementation of a new file system approach as described in this dissertation. It is possible for users to attempt to exploit device and application level diversity manually. A user can direct files to appropriate file systems on top of storage devices that best work with those files' access patterns and the applications that use them. This approach has a number of drawbacks, including a great deal of time and thought required by the user for uniform implementation. With the manual user-based approach, file locations are still tied to the underlying storage device, and large amounts of user intervention are required for nearly every operation.
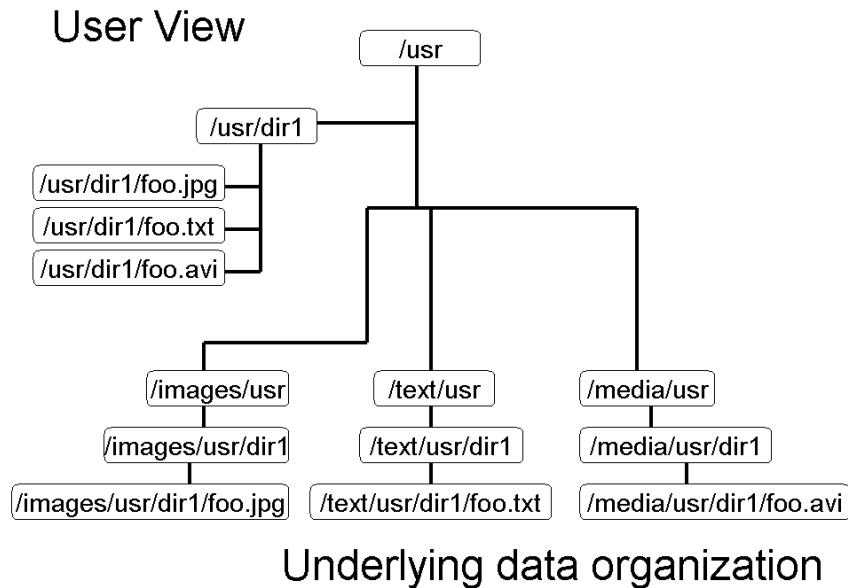
Fig. 5. Example mapping of user namespace based on file contents.

Additionally, it would be possible that rather than providing a combined namespace as UmbrellaFS does, one could rely on soft links to achieve a similar effect. The process could be automated, and files could be placed on appropriate underlying devices while links to the files are displayed to the user in a combined directory. In addition to the lack of transparency in this method, it is likely that the overhead and work involved in maintaining soft links and an additional combined directory structure would meet or exceed that encountered by UmbrellaFS's current implementation.

UmbrellaFS is designed to provide flexibility in storage allocation. Some examples of how this flexibility can be exploited by the user are presented in Chapter V. Much of this flexibility is derived by breaking the one-to-one link from the user

namespace to the underlying storage devices. The flexibility is envisioned to be exploited through a policy module administered by the user or system administrator, the details of which are described below.

System Design

UmbrellaFS functions as a stackable file system [Zadok et al. 2006], residing below VFS and above the underlying file systems. All user interactions with the file systems pass through UmbrellaFS and are directed to the appropriate underlying file system. Some operations, such as open, remove, read, and write which operate on a single file are directed only to the underlying file system on which the file resides. Operations which list the contents of directories such as ls and du must be mapped to all the underlying file systems, and the results must be collated and passed back to the calling operation.

It is possible to provide different functionality by treating the failures on the branches differently when operations such as ls and du are executed on multiple branches or file systems. In one implementation, any failure on any branch is treated as a failure of the entire operation (an OR operation on the failures). In a second implementation, data from successful branches can be passed up, suppressing the failures on some branches (an AND operation on the failures). While more general functions can be employed in reporting failures, these two cases are considered in Chapter V, in two separate applications of UmbrellaFS.

In order to deal with the differences in directories (which may or may not reside on multiple underlying file systems) and files (which should only reside on a single

underlying file system), UmbrellaFS maintains additional data about each file. This data includes values which indicate the branch where the file currently resides as well as the branches where a directory might exist.

The key element of UmbrellaFS is the policy decision module. When any file system operation is invoked by an application, the subsequent system call sends control to the UmbrellaFS. UmbrellaFS then evaluates the file based on the rules provided by the user at file system mount time, and sends the command to the appropriate underlying file system. UmbrellaFS stores these rules in UmbrellaFS's super block. Return values from the underlying file systems are likewise directed back to VFS, which then returns them to the application.

The policy decision module is in many ways similar to a router in a network, directing traffic to the appropriate destination. Many of the policy matching rules (first match, longest name match etc.) resemble the order in which routing decisions are applied when multiple entries in a routing table match an incoming packet. In UmbrellaFS, the file is an analog of a packet and the destination file system is an analog of nexthop. While the current system uses the rules of first match and longest name match, other appropriate rules may be employed in other designs.

An example of the system flow for a call acting on a single underlying file system is presented in Figure 6. The operation leaves the VFS and enters the section of UmbrellaFS that emulates a lower level file system. From this point, UmbrellaFS queries the policy module to determine to which particular underlying file system the operation should be directed. The call then emulates the VFS and calls the lower level

file system call. On return, the return value is passed back up from the underlying file system to the VFS as if UmbrellaFS itself had carried out the call.
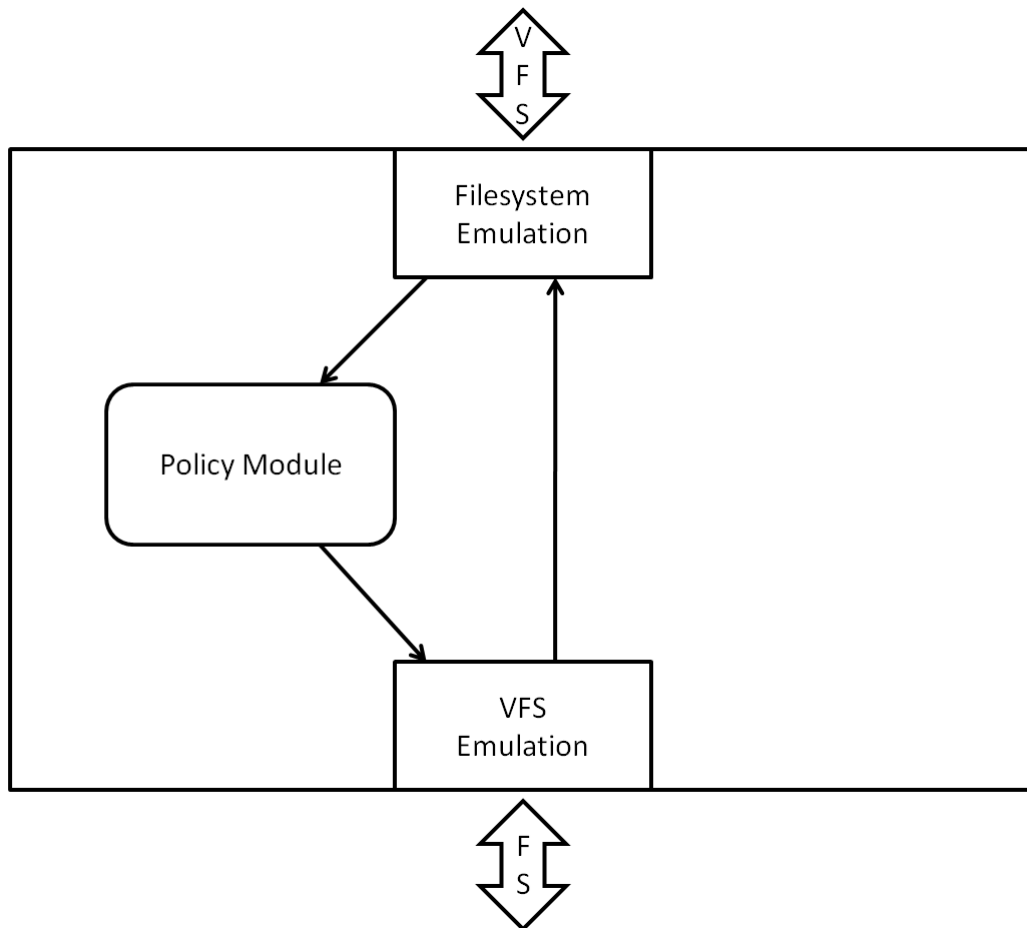

Fig. 6. Interaction of a file system call with the policy module.

Figure 7 shows the system flow for a call that must operate on all the underlying file systems, such as ls. In this case, the policy module is not needed. Rather than being specifically routed to a particular file system, all the underlying file systems receive the

request and then that information is collated by UmbrellaFS before returning it to the VFS.
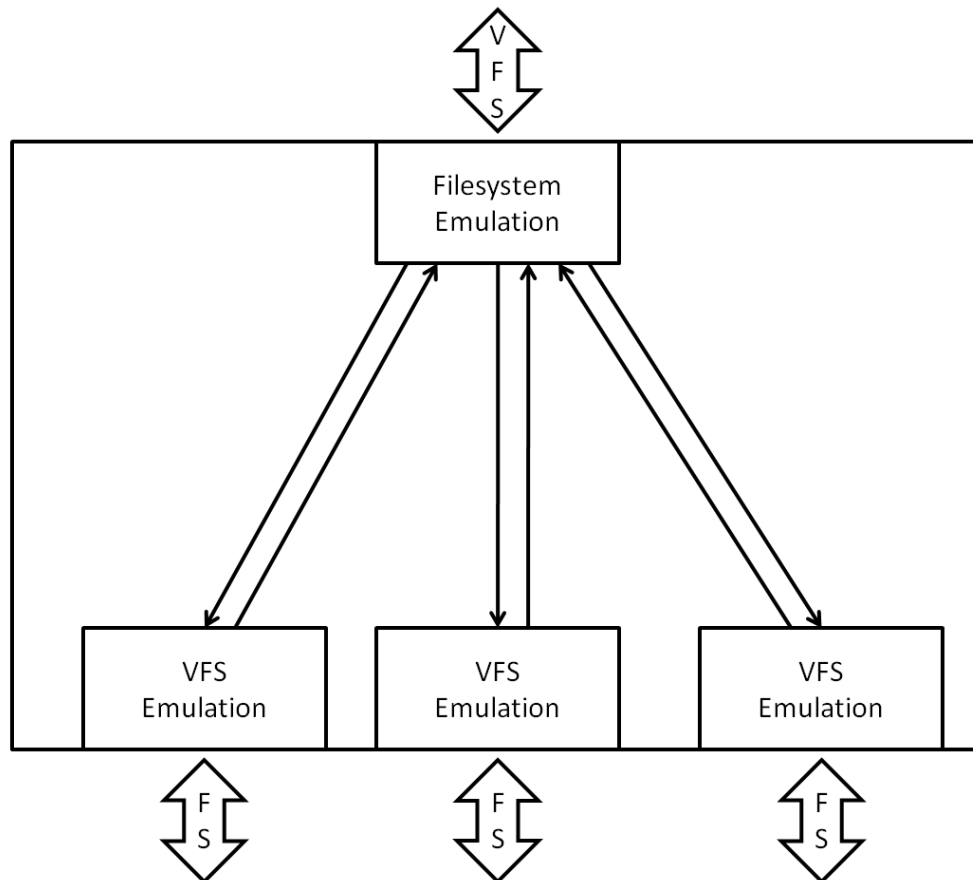
Fig. 7. Operation flow of a call that operates on all the underlying file systems.

The policy module employs rules or policies that determine the allocation of files on different devices. For example, a policy can state that any file larger than 100 kB should be directed to a file system (say /FSRAID) on a RAID5 [Patterson et al. 1988] device. A second policy can decide that Dave's files be stored on a second device. These rules can be based on any attributes of the file, including namespace, size, access

privileges, time of creation, etc. UmbrellaFS directs file system operations to underlying file systems based on the different criteria specified in the policy module. Both namespace rules as well as metadata rules based on inode values can be specified. The rules are evaluated in a two pass system. Metadata rules are evaluated in order, returning the first positive match to the list. The namespace rules reside in the list of metadata rules, and are evaluated when the namespace rule is reached in the list. The different namespace rules are evaluated based on length of match between the file in question and the rule, much like longest prefix matching in routing. In this case however, the namespace rule with the longest overall match, not just the longest prefix is returned and used to determine which underlying file system to use. Should there be additional metadata rules underneath the namespace rules and if there were no matches in the namespace rules, then once the namespace rules return the system continues to evaluate the rest of the rules. Should there be no match whatsoever, the system directs the operation to a default file system.

At the time of file creation, much of the metadata is either unavailable or liable to change shortly into a file's lifetime. Values such as create time can be handled by using the current time of the operation, but values such as file size can be more problematic. As a large file is being copied, it will initially be a small file. If a rule is set up based on file size, this could result in moving a file between underlying file systems soon after creation. In many cases this would be mitigated by the fact that the file that is moved will be less than the full file, however there is an impact that is examined in Chapter V.

Similarly, at file creation, a file would need to be writable, even though it would ultimately be a read-only file. Putting read-only files on a particular file system could therefore involve an overhead of moving files across file systems after file permissions are changed. These overheads can be minimized by employing policies on group and global permissions. Alternately, the mostly-read characteristics can be assessed by other characteristics such as file suffixes ending in .out .exe. While complex rule structures may be needed to adequately describe and define the desired behavior, UmbrellaFS makes these types of rule structures possible.

Figure 8 shows an example system with one metadata and three filename policy rules. These rules are designed to represent the situation described in the introduction of a system with three different types of storage, traditional magnetic storage devices, a smaller capacity flash based device, and another magnetic disk with hardware encryption support. The user, Dave, wants to map most of his personal files to the encrypted storage, read only files to the flash storage, and his video files and all other files to the magnetic storage.

Rule No.	Rule Type	Compare	Value	File system
1	Access type	=	Read only	/fs2, /fs1
2	Filename			

Comparison String	File System
/home/dave/*	/fs3
/home/dave/*.avi	/fs1,/fs3
/*	/fs1

Fig. 8. Sample UmbrellaFS rules.

Rule 1 is based on access type and dictates that all read only files should be located on /fs2, the flash drive. For the purposes of this example, a process such as the previously described planned transition of file permissions is assumed. Rule 2 is based on filenames and is elaborated in the second table. The filename rules state that all of user Dave's files should be placed on the encrypted device, /fs3. The exception is .avi files which should be placed preferably on /fse1 (the magnetic disk) first and then on /fs3. Finally, all other files should be directed to /fs1.

As can be seen from this example, a number of policies may apply to a given file. The specified rules are evaluated in order of specified priority (first rule 1, then rule 2, and then on to other rules if they are present). For example, if user Dave creates a read only .avi file in his home directory, it will be placed on /fs2 since rule 1 is applied before the filename rules are examined. A decision tree or other appropriate data structure can

be used to reason about the effect of the policies on the storage of files on different devices. When evaluating the rules and determining which storage device should hold a file, it is possible that the characteristics that mapped a file to a particular device might change. For example, a file size based rule might be in effect, and a file has grown (or reduced) to the point that the rule structure indicates the file should be on a different file system. These situations are evaluated as they arise, and should a file reside in a file system that is different from the file system that the policy decision module decides it should reside in, that particular file is moved to the appropriate underlying file system. This evaluation is made primarily during file closing (with some exceptions noted below). This movement between native file systems (or underlying devices) may result in performance overheads, which are evaluated later.

Such lazy enforcement of policies on file closing will not be useful with certain rules based on time of access. For example, if one desires a file to be stored on a slow device if it hasn't been accessed in five days, one would need another mechanism to enforce this policy. A periodic policy enforcer may be needed in such cases. This policy enforcer could simply traverse the directory structure, opening and closing each file. This would allow UmbrellaFS to make the appropriate choices for file relocation. The overhead from this type of operation could be minimized by scheduling the policy enforcer to run during off-peak usage times. There is no need for a specific program to move files in the underlying file systems, as this could lead to discrepancies between UmbrellaFS's understanding of the underlying file systems and their actual state. UmbrellaFS depends on the lower level file systems remaining in the state in which it

left them. Attempting to circumvent UmbrellaFS and accessing the lower level file system directly could result in UmbrellaFS not finding underlying files because it is directing operations to file systems that do not contain the files anymore. In general, while UmbrellaFS can be circumvented by accessing the underlying file systems, this should always be avoided to prevent situations like that laid out above. This risk can be diminished by mounting a "dummy" directory over the underlying directory, effectively hiding the directory from the user.

It is possible that the administrator specified policy may not be well suited for the underlying storage systems. For example, the administrator may specify that video files be stored on a 100GB device. If more than 100GB of video files need to be stored in the system, the policy needs to be flexible to use the space on remaining devices that may not be so well suited to video files. In order to allow for these exception cases, the policy can specify a set of choices for locating a given type of files. The policy module tries to place files in the specified preferred order of devices. Hence, in some exception cases, a file system operation such as a read may involve multiple underlying file systems. The impact of this overhead is examined in the experiments.

In these exception cases the writes will be mapped to the next choice of system. If a file were to grow to the point that it could not fit, then the file would have to be moved to the secondary storage system before additional writes are done. This writing is done on the fly as individual actions to the underlying file systems return errors.

In addition to making backup locations like this possible, UmbrellaFS can also explicitly forbid them. If, for example, the rule in question put sensitive files onto an

encrypted drive, one would not want these files to be written to another location if that drive became full. Correct behavior in this case would be to return with an error indicating there is no more room on the device. Whether a backup location is allowed for a given rule is up to the user. In the example above, rules 1 and 2.2 provided backup systems while rules 2.1 and 2.3 only allowed one choice in locating the files.

Inferring beneficial policy rules automatically based on file access behavior in guiding policy decisions may be feasible. This is left to future work. The current framework will be able to accommodate such an approach through the employment of additional metadata characteristics or file tags.

UmbrellaFS can be used to enable different strategies in other layers of the storage stack. Other layers in the storage stack do not have the same information available to them that UmbrellaFS does. Many of the data structures and values are linked together, however, enabling the selective application of other strategies in caching, device drivers, or other layers.

As an example, UmbrellaFS is used to differentiate caching policies as described in Chapter IV. Rather than applying different cache strategies to the entire system, the policies can be targeted specifically to particular devices that benefit from the different strategies. In particular, by marking the pages as they are written to a particular branch within UmbrellaFS, those markings can be interpreted by the caching system to enable alternate methods in that layer.

Implementation

UmbrellaFS is implemented as a loadable module for both the 2.6 and 2.4 Linux kernels. Unionfs 1.0.14 [Wright et al. 2004] was leveraged in the development of UmbrellaFS. Unionfs's ability to combine underlying file systems and representing them in a single directory is an integral part of UmbrellaFS's functionality; however modification of the underlying functions of Unionfs was required in order to accomplish the specific goals of UmbrellaFS. There are no substantial differences in the 2.4 and 2.6 implementations.

The modifications were primarily located in the rule evaluation framework, and the methods for importing the rules into the system at module load time. Some additional minor changes were made in many functions to convert from linear searches of directories to the particular file system targeting used by UmbrellaFS. No kernel code external to UmbrellaFS was modified.

Unionfs combines the contents of underlying file systems into one unified view for the user. UmbrellaFS expands upon this idea by providing the ability to direct files to particular underlying file systems using either namespace rules or other metadata. In addition, UmbrellaFS is designed to separate files into the appropriate underlying storage devices as they are written, instead of combining existing file systems into one. Much of Unionfs's overhead is circumvented in UmbrellaFS because rather than searching through all of the possible directories linearly each time when accessing a file, UmbrellaFS uses the metadata to direct the operation to the appropriate file system. The

inherent overhead from additional system calls is present in both UmbrellaFS and Unionfs.

CHAPTER IV

PAGE CACHE WRITE TECHNIQUES

General Approach

The user and the application perceive the data to be stored as files on the system. From a practical perspective, however, a disk is a collection of blocks. Both the data in files and those files' metadata all reside on a block device. From the device's perspective, one block is much the same as the next. The combination of file systems and device drivers make the translation from the files a user perceives to the blocks that store data on the disk. Thus, when examining the storage system, there are a variety of points where some modifications might have dramatic results on the overall system performance. From the time a user or application calls for a read or write, that call traverses the file system, the page cache, the device driver, and finally the disk itself. In the case of Flash SSDs there is an additional layer in the FTL below the device driver and above the media.

Writes are of particular interest for a variety of reasons. Flash SSDs are greatly impacted by workload mix and have limited write cycles. Additionally, most work has focused on read throughput of the storage system because writes are typically viewed as asynchronous. In a mixed workload or write-heavy system, the writes can quickly become the system bottleneck.

With respect to writes, the data will ultimately traverse all the various layers. Each layer attempts to make logical decisions about placement in order to achieve the

best overall results. Each layer has a limited scope of data it can use to make decisions about when and what to write, and that scope is largely influenced by the layer(s) above it. Thus the page cache is a layer that all writes to block devices transit and has an impact on the decisions that lower levels can make. This is one of the reasons efforts are focused on the page cache and its algorithms for moving dirty pages to the disk.

In this work, the focus is on determination of the order the data should be written out, without altering any policies on how long to retain the data in the system. This experimental constraint allows a fair comparison of the approaches with the currently employed policies in the system. This also insulates the work presented in this chapter from any changes to page cache replacement algorithms or similar aspects of caching that might be developed separately. As the order of writes changes, the data is still bound by the existing page cache heuristics.

In addition to not modifying the various heuristics currently employed by the system, UmbrellaFS limits changes to the page cache behavior to selective parts of the system. This allowed the localization of the policies presented to prevent changing the cache behavior of the entire system. Because of the pervasive nature of the page cache, it was important to ensure that the system was only being changed on the particular part of the system being tested. UmbrellaFS enables the selective application of the particular cache write-back policies under testing such that these changes are only made on the test drive, leaving the rest of the system using traditional page cache behavior.

Various flags are used to determine how pages in the page cache should be treated. In addition to this method which is described later, UmbrellaFS can be used to

enable the use of these other flags. While not a part of the base UmbrellaFS system described in Chapter III, only minor changes were required to expand UmbrellaFS to flag files on particular drives for different treatment. Thus files that the user perceives to be in the same directory can be treated differently by the page cache.

This ability to apply different policies is not limited to write-back treatment in the page cache or assigning files to a certain file system and thus underlying device. UmbrellaFS could conceivably be applied to control policies and affect behavior at various additional layers of the storage stack, such as device drivers. Any situation where UmbrellaFS can affect the behavior of the system is an allowable location to define different policies to be applied to different files, file systems, and devices. For example, the standard cache write-back policy could be used for most of the system, a different policy could be used for some files and a third policy could be used for an additional set of files.

In this particular implementation, when a write goes to a branch that UmbrellaFS has been instructed to flag for application of a different page cache write-back policy, UmbrellaFS enables a flag in that file's inode's address_space structure. This same flag is then checked during write-back, and decisions can be made based on its characteristics. The flag is a simple binary indication that the file should be treated with the different policy. Once a single file in a file system (as determined by having the same superblock) is indicated to be treated with the different policy, all the files on that file system will be treated with the changed policy. This limitation is due to both the file system level nature of UmbrellaFS and the nature of page cache write-back, as the

writeback algorithm traverses a superblock's dirty list to determine what files' pages need to be written to disk.

Write Access Characteristics

In the page cache, the access pattern of data determines decisions such as which pages to remove from the cache, which pages to write to disk, etc. Two broad classifications of access patterns are sequential access and random access.

Sequential data access patterns arise in a variety of situations, and occur in both read and write requests. Playing video or audio files is typically a very sequential read operation. Writing to a log file is an example of a sequential write operation. Both read and write sequential patterns share the property that they typically only visit a particular location in a file once. If that particular page remains in the cache indefinitely, a truly sequential access pattern will not return to it, and it will not help improve page cache hit rates. Sequential data can be thought of as having address locality. That is, accesses are typically clustered based on the address. Temporal locality is common in many types of files that are not sequential, as that data is more likely to be used again close to when it was previously used.

For the purposes of this dissertation, data is considered random if it is not accessed in a sequential pattern. Because the data is not accessed sequentially, it is more likely that an application may need to either write or read the data again. Thus, keeping pages of data that are accessed non-sequentially in the cache can help increase the cache hit rates and the system speed.

In current systems, page caches consider the read accesses to a page in considering how to treat the page in the cache. Pages that are read consistently and repeatedly are held longer in the cache, and pages that are read only a few times are moved more quickly out of the cache. Many cache management algorithms have been proposed that consider the age, frequency and access patterns of data in making decisions about what to retain in the cache and what to evict [Jiang et al. 2005; Jiang and Zhang 2002; Johnson and Shasha 1994; Lee et al. 2001; Megiddo and Modha 2003]. While these algorithms take into account whether or not a page is dirty, they do not consider how the page cache sends writes to the disk.

These write decisions of a page cache have an impact on the performance of the overall system, both from their impact on what pages are in the cache (dirty pages cannot be kicked out of the cache until they are on disk) and in controlling the relative number of dirty pages in the cache. If too many dirty pages are present, then applications will be forced to wait on their write calls, dramatically impacting application performance. Thus, considering the access pattern of write calls makes sense when determining which pages to write in the page cache.

Just as pages that are likely to be accessed again are kept in the page cache longer, it makes sense to defer writes to pages that are likely to be written again. Pages that are unlikely to be written again serve no real positive purpose by remaining dirty in the cache, so it is beneficial to write these pages before writing pages that are more likely to be rewritten. Currently page caches largely ignore this distinction and write pages in an order based on when the page was first written. Categorizing these pages

based on their write patterns would help leverage the same ideas used in page cache eviction to improve rewriting and overall write performance of the system. For example, in Linux, when the 10% dirty threshold is reached, could performance be improved by taking the write characteristics of the page into account, rather than simply evicting the pages in the FIFO order as done currently?

Particularly with the advent of Flash SSDs and their differences from magnetic disks with respect to writing, improving write performance has even more opportunity to improve overall system performance. A reduction in overwrites to a Flash SSD has a more dramatic effect than the same reduction in overwriting on a magnetic disk. The erase cycles in a Flash SSD are a large source of latency, and separating sequential and non-sequential data in the page cache write algorithm could help reduce erase cycles.

Multiple Queues

Separating the writes into multiple queues based on their access patterns provides the ability to discriminate between various pages that would be better to write immediately and those that would benefit the system by staying dirty in the page cache longer. It is possible to separate the writes into multiple queues based on their write access patterns. Since there are two obvious distinctive write patterns, utilizing two queues, one for sequential and one for non-sequential access patterns, this strategy is explored in this dissertation. While there are additional possible classifications of the access patterns, such as a looping sequential access pattern, detection of many of these other patterns requires more state to be maintained and more processing to be done to detect them.

The system described in this work utilizes two queues, one for sequentially written files and one for non-sequentially written files. When writing is required, sequentially written files are written before files from the non-sequentially written queue. This way it might be possible to reduce repeated writes to the same files, as the random files will stay in the cache longer allowing more write hits on dirty cache pages, as well as clearing up space in the cache from the sequential accesses that are unlikely to be written again before eviction.

In order to provide sequential detection, the number of times the pages within inodes are written is tracked. If a page is written more times than a certain threshold, it is classified as non-sequential or random. Otherwise it remains sequential. Rather than sequential detection, this strategy accomplishes "non-sequential detection," although alternate implementations are certainly possible.

Sorting

Magnetic drives are well known to have seek times and rotational latency penalties. Moving parts must get into the proper position in order to write to or read from the disk. Numerous strategies have been employed to improve this performance such as elevator scans and request merging. Because most of these concepts are implemented in the lower layers of the storage stack, decisions higher up can have an impact on the efficacy of these algorithms. By improving the order that writes leave the page cache, the overall performance can be improved with the help of these lower level algorithms.

Many flash drives organize the data internally in terms of larger erase blocks (even though writes may be done at a smaller size) and such organizations make it more efficient to write data in larger sizes. These issues drive the motivation in particular for ordering the writes leaving the page cache.

 Algorithms in the scheduler and device drivers sort the writes in order to optimize the writing process on the underlying media. Despite this reordering, the initial order that is chosen in the page cache for writes clearly affects the ultimate order of writes on the device. Hold times and buffer sizes are issues at the various lower levels, so those layers are limited in their effectiveness by the order of writes coming from the page cache.

On the other hand, historically these write order decisions have been made at the lowest levels for very solid reasons. The lower levels have much better information about the ultimate block addresses of data than higher up the stack. The page cache does not know precisely where on the disk a page resides. The scheduler and device driver can sort and reorder the pages more cheaply than the page cache because they are working with smaller sets.

The page cache has some benefits that are not present in the lower levels. It has access to all the cached data in the system, so it can make decisions collectively about more data than the lower levels are given at any given time. Despite not having precise block information, the page cache can make inferences about the relative locations of pages and files on the disk.

This research studies the effectiveness of sorting the blocks being written out at the page cache. In particular, sorting the files that are slated to be written out before they are passed out of the page cache for writing is examined. This gives the system the opportunity to reduce the frequency of SSD boundary crossings in Flash SSDs and thus improve overall performance. While most sorting algorithms have time complexities of at least $O(n \log n)$, the write operation itself is so long that this overhead has not been seen to be significant in tests involving tens of thousands of files.

Example Scenario

As an example, assume there are 7 files, each of which has a single 4k block. These files can be described by their inode numbers, which for the sake of clarity are labeled with the integers from 1 to 7. These files are written in the order of 1, 5, 3, 6, 7, 4, 2. Next the even numbered files are overwritten with new data multiple times. In a normal Linux system, the writeout algorithms would traverse the superblock's dirty list and send the blocks of these inodes to disk. In the case of this example, let us assume that writing out 3 files will satisfy the present needs for writing.

In a traditional Linux system, the write-back would proceed as shown in Figure 9. The files (1, 5, and 3) are written in a FIFO manner and the remaining inodes remain on the superblock's dirty list. In this case, that leaves inodes 6, 7, 4, and 2 as dirty.
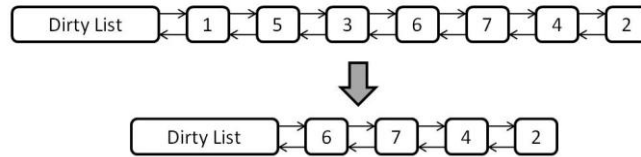
Fig. 9. Traditional Linux.

In the multiple queues implementation, the inodes would first be put into sublists based on whether they have been detected as having multiple writes. This is shown in Figure 10. Inodes 6, 4, and 2 are placed on the nonsequential list and 1, 5, 3, and 7 are placed on the sequential list. Inodes 1, 5, and 3 are written out in that order, and the remaining inodes are returned to the superblock's dirty list.
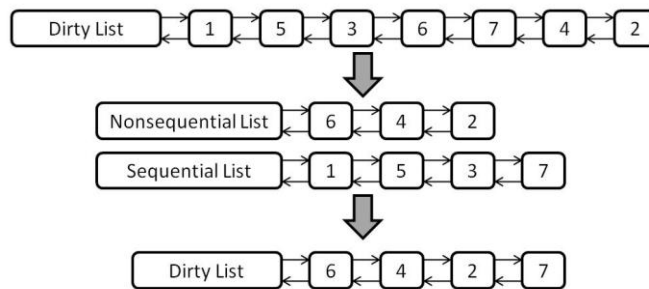

Fig. 10. 2 Queues.

The sorting implementation is shown in Figure 11. The inodes are placed on a temporary list in sorted order, and that list is traversed for write outs. Thus inodes 1, 2, and 3 are written out and inodes 4, 5, 6 and 7 are returned to the dirty list.
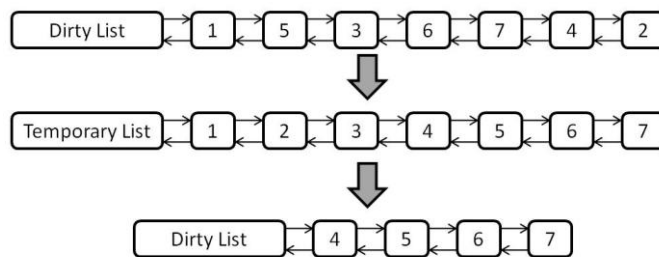
Fig. 11.   Sorting.

When both multiple queues and sorting are used, the method proceeds much as in the multiple queue implementation, but as the inodes are placed on a sublist based on their characteristics, they are placed in sorted order.   Thus the nonsequential list is ordered 2, 4, 6 and the sequential list is ordered 1, 3, 5, 7. After writes are completed, 2, 4, 6, and 7 are returned to the dirty list, as is shown in Figure 12.
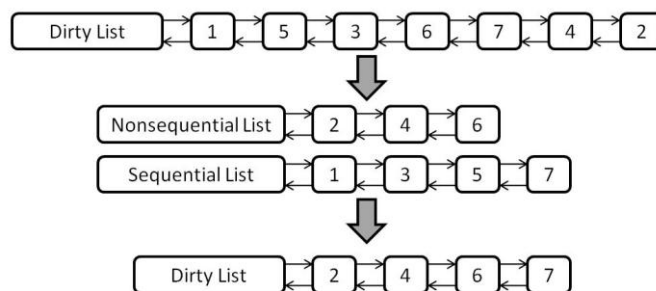

Fig. 12. Sorting and multiple queues.

Implementation

This implementation of both the multiple queues and sorting techniques to improve system performance is based on a Linux platform. As a kernel based solution, an actual working implementation of the modifications in the page cache has been created.

Linux attempts to balance the amount of dirty data and length of time it remains in the cache using two thresholds and a timer. One threshold is for background writes and one is for foreground writes, where a process is stopped until some of its writes complete. These thresholds are based on how many dirty pages are in the page cache as a percentage of the total size of the cache. The background writeout threshold was originally 10% in the Linux kernel i.e., when the number of dirty pages exceed 10% of the total page cache size, a background write process is triggered to offload the dirty pages to disk. When the number of dirty pages reaches 40% of the cache threshold, processes would not be allowed to proceed until their writes completed. When the background process keeps the dirty pages below the 40% threshold, the applications complete the writes to page cache and do not see a disk delay for writes. Due to the increase in memory size, where desktop computers now often have 2 to 4 gigabytes of RAM, those thresholds have been lowered to 5% and 10%, respectively.

In addition to these thresholds, Linux policies establish a maximum time a page can remain dirty. If a dirty page is still in the page cache after 30 seconds from its initial write, then background threads wake up and write the changed files to disk. Both this as well as the background and foreground thresholds attempt to allow enough writes to

collect that the disk will be utilized well and to ensure that changes are written to disk in a relatively timely manner. This set of policies minimizes the chance that a system failure will be catastrophic from a data loss standpoint.

These rules and heuristics control the aggregate write behavior from page cache to the disk. Relatively little effort has been directed towards which pages should be written when it is determined that something should be written out. For example, the Linux kernel does not employ any particularly complicated strategy when making this decision. Inodes and their data are written in the order that they reside on a doubly linked list. When a file is written, it is marked dirty and placed on the list of dirty inodes. When that data needs to go to disk, either because it has been 30 seconds, or the page cache is above one of the thresholds, inodes and their data are written out in a first-in-first-out (FIFO) order. Inodes are not repositioned if written multiple times, so the first file to be written will go to the disk driver first. This implementation has low computational overhead, and since writes are typically asynchronous, the exact order that pages reach the disk is traditionally considered to be relatively unimportant to full system performance.

In the page cache there is a doubly linked list of all the inodes of dirty files. However, it is not known precisely where these files reside on disk. The underlying mapping is not easily accessible from the page cache. However, the inode numbers for the various files are known. Since most file systems use inode numbers in a relatively sequential manner, it is likely that that by ordering files according to their inode number, a more sequential access pattern to the disk might result. This is a simple hueristic that

is seen to be quite effective in tests. Aged file systems might be less likely to have particularly sequential use of inodes. In situations such as this, it will be necessary to employ techniques based on the block addresses of the files. This is left as future work.

A selection sort algorithm is used to take the inodes off the superblock dirty list and then place them in a sequential order for write-out. This algorithm is triggered every time a background writing thread wakes up to do write-out, or when an application is forced to devote its processor time to disk writes. If inode write-out is prematurely terminated due to the write limit in a single pass, the remaining dirty inodes are returned to the superblock dirty list, although now in a sorted order. The original FIFO order is neither maintained nor restored after the sorting operation. If additional inodes are added to the dirty list, they are added in the traditional manner at the end of the list and then sorted once again when write-out begins.

Although selection sort has a complexity of $O(n^2)$, in tests involving up to tens of thousands of separate files being affected by small writes significant overhead was not observed. Some improvement might be achieved by using a faster $O(n \log n)$ sorting algorithm, although in practice writing to disk takes sufficient time that the initial sorting complexity is a small portion of the total time spent doing device I/O.

To keep track of write characteristics, unused bits in the page flags were used to mark pages as they are written. Two new page flags were declared, a written bit and a sticky bit. This allows the counting of up to four writes. When a page is written multiple times, the flags are modified in order to increase its counter. Once the counter reaches a certain threshold, the file is considered to have repeated writes. Heuristically it

was determined that after 4 total writes to the same page in a file that file should be marked for inclusion in the repeated writes queue. A count of 4 writes to an individual page helped prevent too rapid escalation of a file to repeated write status, as well as fitting in two bits in the page structure in the cache.

Files are marked via the address_space mapping in their inode. The upper bits of the address_space mapping flags are not used, and thus can be overloaded for this implementation. Additionally, the inode's address_space is already available in the function that is doing the writing to individual pages. Thus the flags indicating the number of times an inode is written can be checked and set, and then mark the inode's address_space appropriately without the need to reference additional objects that are not already present in the function. The uppermost bit of the address_space mapping flags is marked to indicate that the inode in question should be treated with the two list strategy. The next bit is marked after any page within an inode receives its 4th write. In this initial implementation, the examination is done on a per page basis because it is units of pages that are in the page cache and are actually written to disk. Because of the difficulties in accessing the particular pages when doing actual writes to disk, files are treated as sequential or random based on whether a page is written repeatedly or not. Although this may lead some files to be misclassified, the results (shown later) indicate that this policy improves performance on the whole.

When the system decides to write dirty pages from the page cache to disk, the files are taken from the superblock dirty list and placed onto one of two lists depending on whether the file in question has been flagged as being sequential or non-sequential.

The sequential queue is then written first, and if the system still desires to write more pages, the non-sequential list is written to disk. Any files that remain dirty are returned to the superblock dirty list. When testing both the techniques of multiple queues and sorting simultaneously, files are placed into the two queues in a sorted order. That is, rather than keeping them in the order they were originally on the superblock dirty list, the files are placed into the two new lists using insertion sort.

CHAPTER V

RESULTS

UmbrellaFS Results

Benchmark tests for UmbrellaFS were run on a Dell Optiplex GX620 with an Intel 3.2 GHz Pentium D processor and 2 GB of RAM. Tests were run using a Red Hat Fedora Core 3 Linux 2.6.9 kernel as well as a Debian 2.4.27 kernel. For tests the kernel only used 250 MB of RAM in order to prevent caching from skewing the results. The system disk was a Samsung 7200 RPM 250 GB SATA hard drive, while the tests were run on Seagate Cheetah 10k.7 73 GB SCSI hard drives. The SCSI hard drives were connected to the system via Adaptec 29320A SCSI controllers. Ext2 was used as the file system on all hard drives. The lower level file systems were remounted between each test to clear file system caches.

A variety of macrobenchmarks and microbenchmarks were chosen to explore the characteristics of UmbrellaFS. In addition to the benchmarks examined in [Traeger et al. 2008] such as Bonnie++ [Coker 2001], Postmark [Katcher 1997], and compilation of source code, particular microbenchmarks were developed in order to examine some of the potential issues in UmbrellaFS. Each test was run at least 3 times, and the averages are provided. Ninety-five % confidence intervals were calculated using the Student-t distribution, and these confidence intervals also appear on the graphs. The half-widths of the intervals are less than 5% of the mean in most cases, and exceptions are noted in their particular sections.

UmbrellaFS's overhead was evaluated in a number of different situations. Raw device throughput, CPU and I/O intensive loads were examined. All tests were performed with exclusively inode rules as well as exclusively filename rules. When inode based rules were used, the worst case scenario (no match on the rules until the final rule) was used. Similarly, with the filename rules, rules were chosen which had at least some match to the files used so that as much time as possible would be spent comparing various string matches.

Basic overhead was examined when UmbrellaFS was mounted over a single underlying file system. Additionally, overhead was examined when UmbrellaFS resided over multiple underlying file systems, both when data was confined to a single branch, and when data was distributed between branches. The multiple branch results are not included, but do not differ significantly from the results shown.

Bonnie++

Bonnie++ is a benchmark which tests sequential accesses to a single file, or multiple files if the working size is larger than 1 GB. A working size of 2 GB was used for Bonnie++, so Bonnie wrote two 1 GB files. The results of the Bonnie++ benchmark are shown in Figure 13.
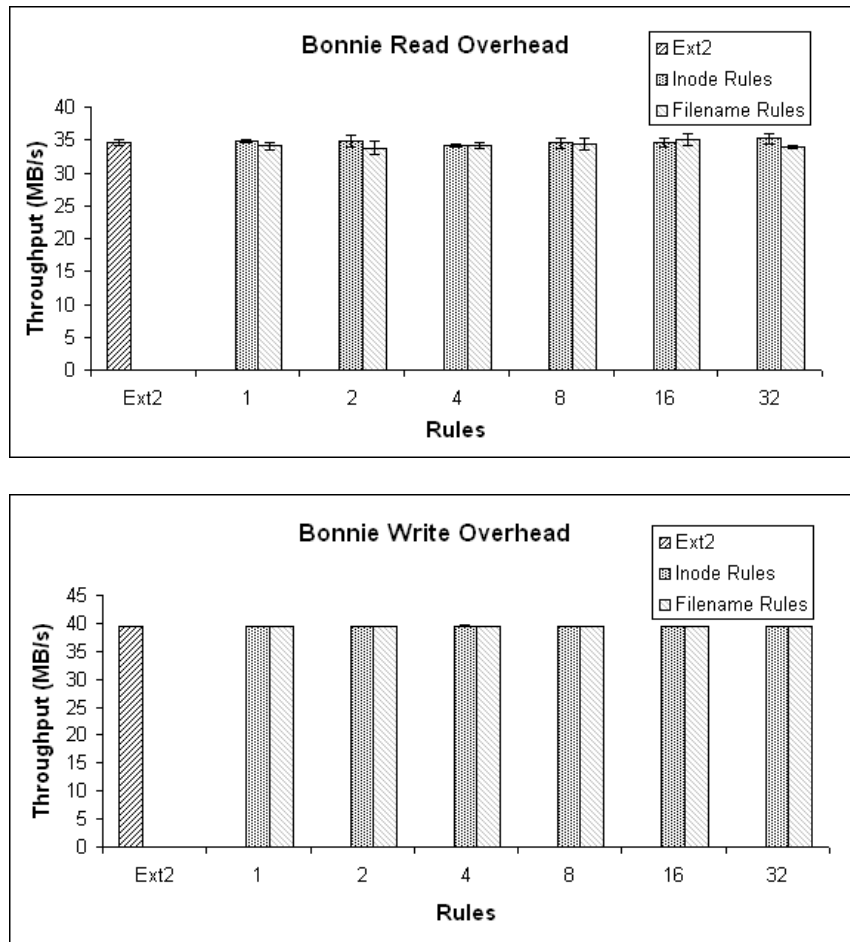
Fig. 13.   (a) Throughput in Bonnie++ benchmark for sequential reads.   (b) Throughput in Bonnie++ benchmark for sequential writes.

As can be seen in Figures 13(a) and 13(b), UmbrellaFS imposes no discernable overhead in large sequential reads and writes. The read benchmark in particular was rather noisy, as can be seen from the confidence intervals. The half widths of the intervals in the read test are all less than 10% of the mean, but the means themselves are often within the confidence intervals of other points in the graph. The confidence intervals for the write overhead are several orders of magnitude smaller, and the means

are very close to the base ext2 case. From all of this one can infer that there is not a large overhead associated with large reads and writes.

OpenSSH

Compiling OpenSSH [OpenBSD 2007] is a CPU-intensive benchmark [Wright et al. 2004]. A vanilla copy of the OpenSSH 4.7p1 source code was used for these tests. The benchmark itself consists of a typical compilation and installation of OpenSSH, first by running configuration tests, compiling the OpenSSH 4.7p1 code, and finally installing OpenSSH. This is the typical series of configure, make, and make install to compile and install a program. This test involves numerous reads, writes, and other system operations, and helps expose UmbrellaFS's potential impact on typical users. The benchmark was timed, and the results are shown in the amount of time spent in user mode, kernel mode, and wait time, which typically corresponds to I/O, although wait time could also be impacted by many other factors in the system.
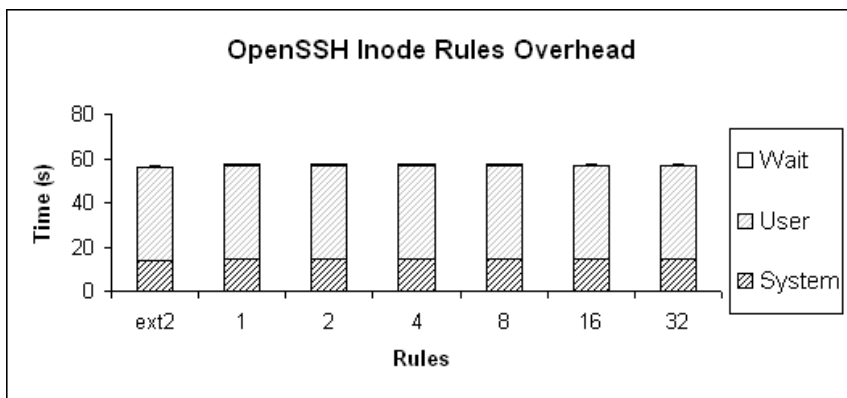

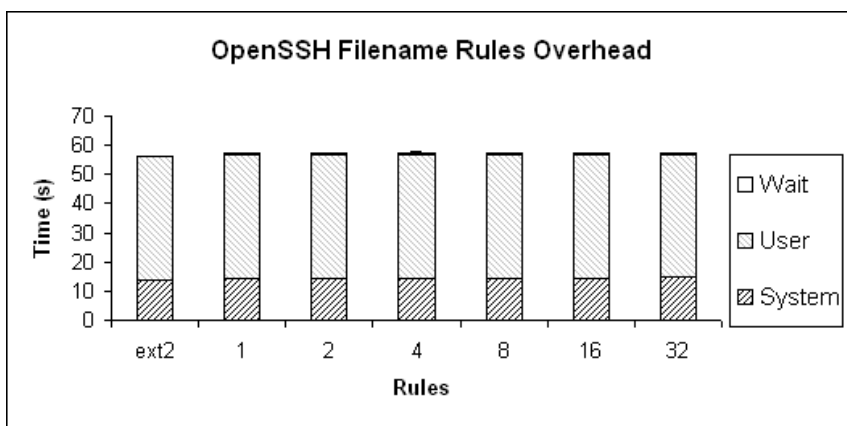Fig. 14. OpenSSH overhead with inode based rules.

Fig. 15.   OpenSSH overhead with filename based rules.

Figures 14 and 15 show the overhead of the OpenSSH benchmark for inode and filename based rules. The confidence intervals shown are for overall execution time. In general, the half widths of the confidence intervals were within 1% of the mean, for overall, system, and user times. Wait time had a very large confidence interval. Given the relatively small amount of wait time, this did not have a large effect on the overall execution.

With both inode and file based rules, the predominant overhead is in the kernel execution time, as is to be expected. UmbrellaFS requires an extra system call for every file system operation, since instead of accessing the file systems directly, VFS accesses an UmbrellaFS function. The average kernel execution time in the inode rule test was actually lower for 32 rules than for a single rule, and in the filename rules the overhead was less than 1% higher than the single rule case, indicating good scaling properties for the number of rules.

Postmark

Postmark is an I/O intensive benchmark designed to simulate the operation of an e-mail server.   In the Postmark tests, Postmark version 1.5 was used to create 12,500 files between 8 kB and 64 kB in 200 subdirectories and then performed 25,000 transactions.   The block size was 512 bytes, with the default operation ratios and unbuffered I/O.

Figure 16 shows the results of the Postmark tests. Both types of rules showed an overhead on a straight Ext2 partition, typically less than 1% but at no time more than 3%.
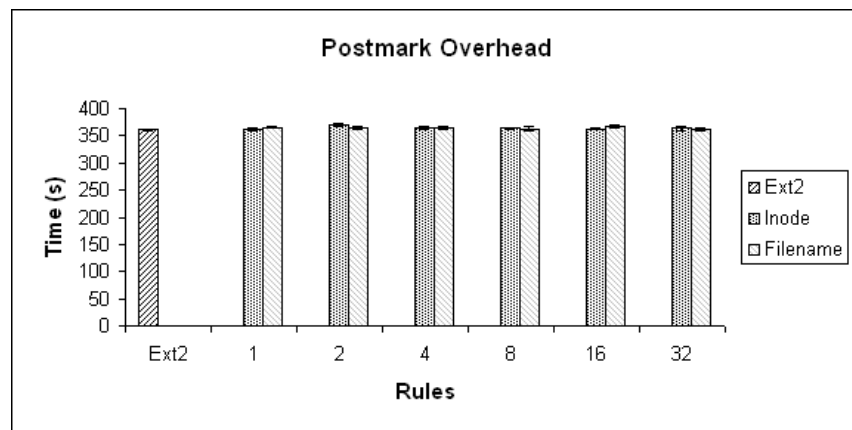
Fig. 16.   Postmark Overhead.

In the Postmark evaluations, UmbrellaFS showed a consistent overhead over the vanilla Ext2 results. Since Postmark is a benchmark that spends almost all of its time reading and writing to random files, this is not surprising. That both CPU intensive and random I/O intensive benchmarks show less than 3% overhead across the board for

UmbrellaFS indicates that in most user applications, the overhead of simply installing UmbrellaFS will not be prohibitively high.

Overflow

Because the Postmark benchmark had the highest overhead with UmbrellaFS, Postmark was also used to determine the overhead when one file system fills up and another is used in a backup capacity. A single inode-based rule and two branches were used. One file system was filled with data, while the other was empty at the beginning of the test. The rule indicated that writes should go to the full file system first, and then the empty file system. Figure 17 shows the results, compared with vanilla Ext2 and a single inode-based rule on one file system. The overflow system had an execution time very slightly lower than the single rule test, although given the confidence intervals it is not an unreasonable value. The overflow situation showed 2.6% overhead over vanilla Ext2.

As can be seen from Figure 17, simply being forced to write to another drive when the first drive is full imposes no meaningful overhead beyond that imposed by UmbrellaFS in general. In this situation, there was no need to copy files across file systems, since the primary file system began the test full. The overhead of being forced to copy files is evaluated later.
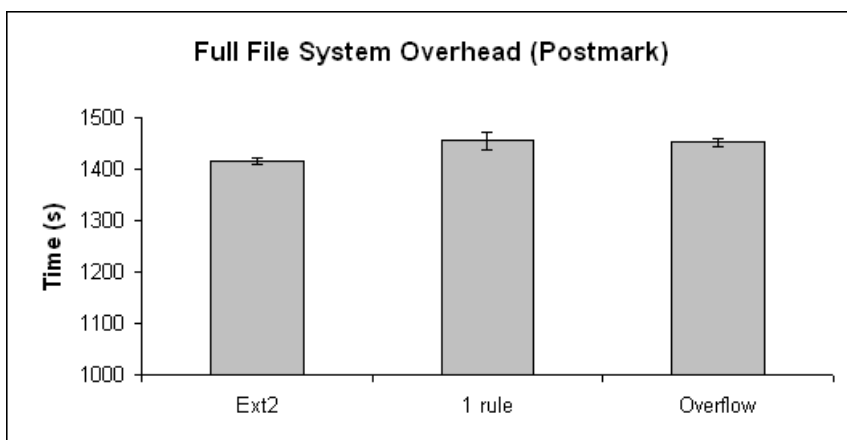
Fig. 17.   Postmark overhead with full file system.

Rewriting Overhead

Situations may arise where a file is moved back and forth between two branches on separate drives.   While careful application of rules can reduce the risk of this thrashing, it is still conceivable that the properties of a file will hover near the boundary between two rules, requiring multiple rewrites of a file as the characteristics change.   In order to examine this situation, a micro benchmark was used which took a set of 35,000 files, appended to each file, and then truncated the file by the same amount, and finally appended again to the file in order to keep it at the larger size.   This was repeated 3 times, with the middle set of append, truncate, append taking the file across a rule boundary.   The system was set so that files larger than 8 kB were on one file system, and those smaller were on a different file system.   Initially files were 6.5 kB in size, and all appends and truncates were 1 kB in size.   The results of appends and truncates which moved the file size between 7.5 kB and 8.5 kB are shown in Figure 18.   The appends and

truncates when not on a rule boundary resulted in overhead in line with that experienced

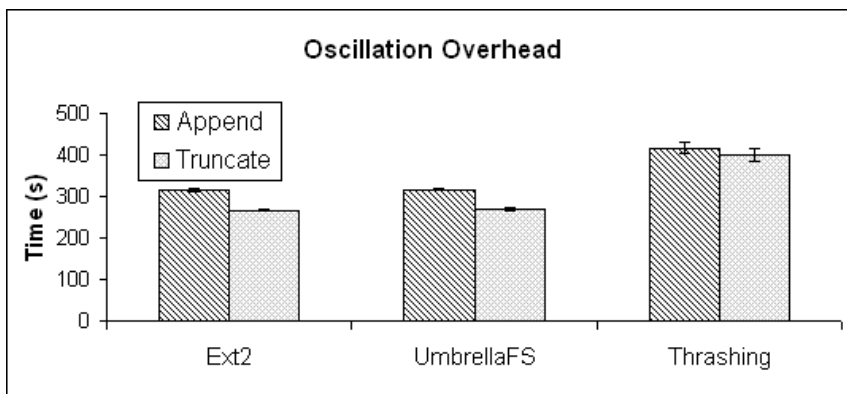in previous benchmarks, and those results are not shown.

Fig. 18. File size oscillation between 7.5 and 8.5 kB.

When a 7.5 kB file was appended with 1 kB of additional data and there was a

rule indicating that files larger than 8 kB should be stored on one drive and smaller than

8 kB should be stored on another drive, significant overhead was seen. This is expected,

since in a typical append of this size, only an append is done. To move the file, a new

file must be created on the other drive, and the full 8.5 kB must be written, rather than

just the 1 kB append. The original file must also be deleted. Likewise, on a truncate the

file is moved back to the original file system. Appends showed an overhead of 32% on

ext2 and 31% over UmbrellaFS without the file movement. Truncates showed an

overhead of 49% and 48% respectively. The severity of these overheads demonstrates

the importance of careful rule selection. It is noted that rules based on thresholds could

be susceptible to this overhead if the workload causes the files to go back and forth

across the thresholds, and hysteresis of some sort is being considered in order to reduce this type of thrashing.

Example UmbrellaFS Scenarios

This section demonstrates the potential utility of UmbrellaFS in a number of example scenarios. The first example consists of a system with diverse devices. The second example demonstrates a system where different native file systems are employed below UmbrellaFS. The third example considers different failure semantics and shows the possibility of a Coda [Kistler and Satyanarayanan 1992] like disconnected operation.

Diverse Devices

In the first example scenario, consider a system consisting of a Flash drive and a RAID array. Samsung's 32 GB Flash drive (model MCBOE32G8APR-0XA) and 5 of the Seagate Cheetah 10k.7 drives organized as a 4+P software RAID5 array were employed. The characteristics of these two devices are shown below in Figures 19(a) and 19(b).

The Flash drive has superior read performance at request sizes up to 2 MB and better write performance at smaller write sizes up to 64 kB. The RAID array has higher read and write performance at larger request sizes. It is noted that the performance gap of writes between the flash drive and the RAID arrays is significant (up to a factor of 4) at larger file sizes.
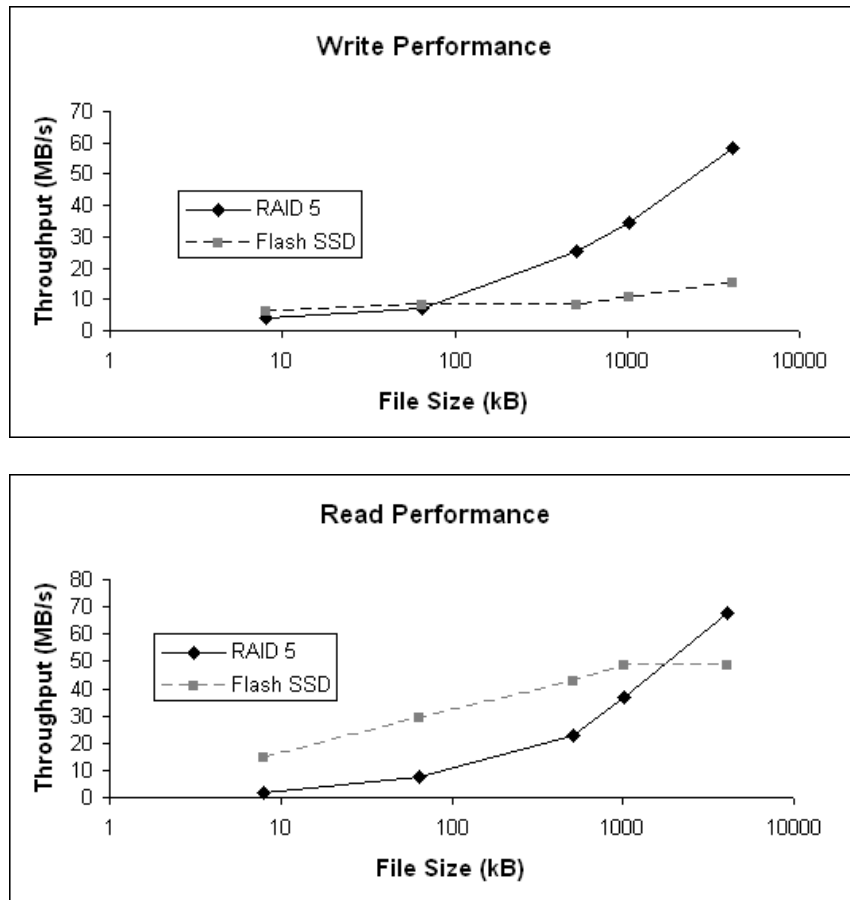
Fig. 19.   (a) RAID 5 and flash SSD write performance.   (b) RAID 5 and flash SSD read performance.

In order to mitigate the impact of limited number of write cycles of flash drives and the write performance gap at large file sizes, it was decided to allocate read-only files and executable files on the flash drive while placing the remaining files on the RAID array. Postmark was modified to emulate this situation.   Two file sets were created. One file set was only read, while the other file set was both read and written to. The policy made this decision based on filenames in this example; however in a real system the file permissions could just as easily be used. File sizes in this simulation

were between 8 kB and 1 MB. The performance of such a policy on the example system
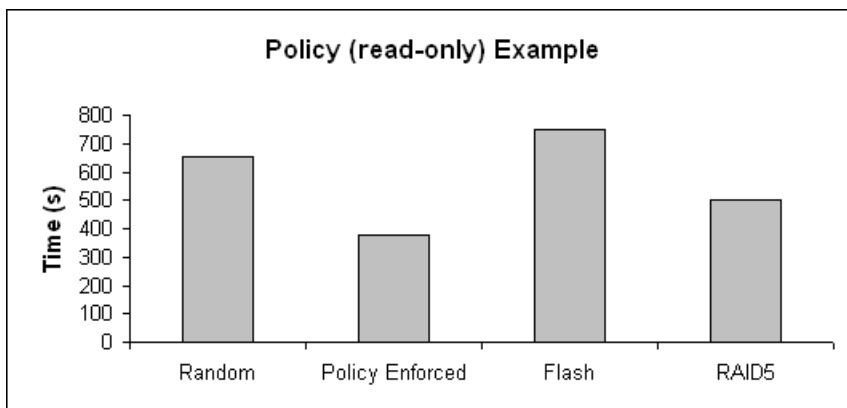
is shown in Figure 20.

Fig. 20. Diverse devices policy rule example.

The results of this policy are compared against the policy of randomly assigning

files across the two types of devices, as well as placing all the files on either the flash

drive or the RAID array. The policy took 44% less time than a random distribution, and

51% and 26% less time than flash and RAID 5, respectively. The results show that by

keeping writes on the RAID 5 array and heavily utilizing the Flash device for reads, the

overall performance is improved. In the random distribution example, the large writes to

the slower flash device impacted performance. In this example, the flash drive did not

fill up, which could have forced overflow to the RAID array or vice-versa. Even had

this occurred, however, because there would not be a need to move the files, the system

would not exhibit the overhead seen in the "Rewriting Overhead" section of Chapter III,

but rather the performance would have reverted to the RAID 5 curves in Figures 19(a)

and 19(b). Performance improvements are dependent on many factors, viz., the relative performance of the devices and the relative loads on the different devices, localities of loads and other factors and that performance gains from policy decisions may not be universal [Shenoy et al. 1998].

More importantly, the policy of allocating read-only and executable files to flash drive through a system level policy improves the lifetime of the flash drive since writes are incurred only once when the files are initially written to the flash drive. Second, if the user or administrator deems that small files can also be effectively placed on flash drives (because of RAID array's small-write cost), it is easy to modify or augment the system policy to make this change.

Diverse File Systems

Another example scenario was set up similar to the previous situation. In this case, the read only file set consisted of 50,000 small files, with sizes ranging from 1 kB to 5 kB, rather than the large files represented in the previous example. The read/write file set still consisted of files between 8 kB and 1 MB. The performance of this situation was examined in terms of the time to create the files as well as the time to run transactions on them (reads in the case of the 50,000 small files, reads, appends, creations, and deletions for the large files). The small files were placed on the Flash drive as before, but this time the file system on the Flash drive was Ext2 for one set of runs and NILFS [Yoshiji et al. 2008], a Linux log-structured file system, for another. The other set of runs simply placed all the files on the RAID 5 system. The results of the test are shown in Figure 21.
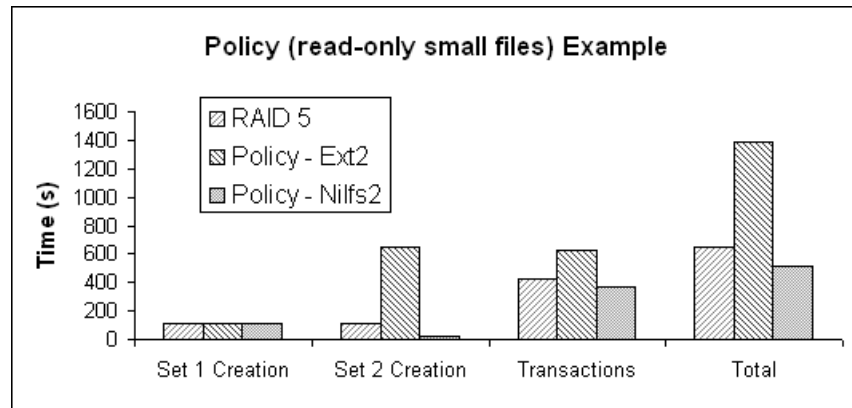
Fig. 21.   Diverse file systems policy rule example.

As can be seen from the results, the Flash device performs poorly with respect to file creation with numerous small files when it uses the Ext2 file system. When using NILFS, however, the Flash device took 79% less time to create the files than the RAID 5 array, and over 96% less time than the same Flash device using Ext2. Transactions took 11% less time with the policy based storage and NILFS on the Flash device and 40% less time than the same policy-based storage with Ext2 on the Flash device. These results, particularly in combination with the previous policy-based storage results clearly demonstrate how different workloads need different policies. In particular, these results demonstrate that different file systems can provide a drastic difference to results for different workloads. As this example shows, UmbrellaFS makes matching particular file systems to devices and workloads possible within a single namespace.

Encryption

In order to examine a situation with multiple types of file systems, consider the example of a system with 3 types of storage devices, a magnetic disk, a magnetic disk with encryption, and a flash drive.

Difficulties were encountered obtaining Linux drivers for full disk encryption drives such as the Seagate Momentus 5400 FDE.2. So rather than relying on a hardware encryption solution, EncFS [Gough 2006], an open source encrypted file system that makes use of the FUSE [Szeredi 2007] library to run in user space, was used in a software-based emulation. The blowfish algorithm was used to encrypt files with a 160 bit key. While this does not function exactly as a hardware-based encrypting drive would, the overhead from encryption is present, as well as demonstrating the functionality of UmbrellaFS working across multiple types of underlying file systems.

The same modified Postmark benchmark as the previous example was used in this situation. Three drives were used, two of which were the Seagate Cheetah drives, as well as the Samsung Flash drive. The numbers in this example and the previous one are not intended to be directly compared.

Figure 22 shows the results from a policy where only the 10% files placed on the "encrypted" drive were encrypted compared to a policy where the files are distributed the same way between the different drives, but all three file systems were encrypted. As is to be expected, when only a portion of the files encounter the overhead of encryption and decryption, the benchmark takes much less time to complete than when all files need to be encrypted before being stored and decrypted before being read. This example

demonstrates how appropriate use of rules can reduce the burden of particular file placement on underlying drives on the user. It also shows the ability of UmbrellaFS to function across multiple different underlying file systems.
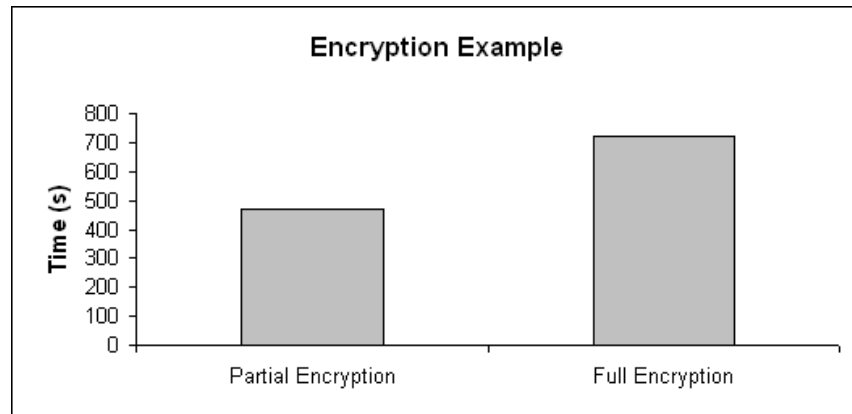

Fig. 22. Encryption example.

Disconnected Operation

CMU's Coda system achieves disconnected operation in a networked storage situation by caching recently used files on the local machine. When the machine is disconnected, operation can continue on those local files, and then when reconnected, the copies on the network are updated to match the locally cached copies. With appropriate use of rules, UmbrellaFS can achieve a somewhat similar type of operation.

In this situation, assume that the user has a local machine and has mounted an NFS partition as well. A drive on the local machine and the NFS partition can be used to achieve disconnected operation. By setting up a rule based on access time, all files which are accessed today can be moved from the NFS partition to the local machine.

If the NFS server should become unavailable, either due to a network outage, or simply because the user's machine has been disconnected from the network (in the case of say, a laptop that the user is taking home for the day), then operations that attempt to access the directory that UmbrellaFS is maintaining will return an error. If rather than an OR based evaluation of errors (where any error in any of the underlying branches returns up an error to the user), an AND based evaluation is utilized, it is possible to hide this error from the user and take appropriate action to return the available information. The choice of between OR and AND based evaluation of errors can be made when the module is loaded, the same way the rules themselves can be specified.

When an error of this type is encountered, a feature of Unionfs is used. That feature is the ability to add and remove underlying branches on the fly. By calling an appropriate ioctl, the branch that returned the error from UmbrellaFS cab be removed so that future actions will not attempt to access that branch. This functions in the same manner as in Unionfs, although in addition to the work that is normally done, the rules must be updated. If not, then an access that would normally go to the (now disconnected) device would cause problems such as accessing indices that are not present, etc. So when a branch is removed, the rules in the system are analyzed, and if any rule's primary file system that it directs to should become unavailable, that rule is flagged as inactive. Inactive rules are then simply skipped by the rule evaluation mechanism. When evaluating the placement of a file, if no rule applies to a file because of a rule deactivation, then just as if there is no rule in the system that applies to that file, the file is placed on a default file system.

When the NFS partition is reconnected the same Unionfs feature is used to add branches on the fly. Again the rules must be evaluated, and if an inactive rule's primary branch has returned, that rule is reactivated so that it again directs files to the newly returned drive. In this situation, however, there are files that have been moved to the local machine and need to be migrated back to the network storage. In order to accomplish this, a background program is utilized. Since the evaluation of a file and its correctness of location is done on open and close, files cannot move back to the networked storage without assistance. The background program simply opens and closes files, so that the evaluation can take place. Upon closing, the most recent access time is evaluated, and if the file has not been accessed recently, then it is moved back to the network storage device. Since the program does not actually read the file, the file's access time is unaffected and can be properly examined.

By using these elements of the UmbrellaFS system, disconnected operation similar to that provided by Coda can be achieved, without requiring a fully new file system. The examples presented in this paper are not intended to be comprehensive, but rather an indication of the types of things that can be achieved with an UmbrellaFS type system.

<div align="center">

Page Cache Evaluation

</div>

All the benchmark tests for the sorting and multiple queues implementations were run on a Dell Optiplex GX620 with an Intel 3.2 GHz Pentium D processor and 2 GB of RAM. Tests were run using a Red Hat Fedora Core 9 Linux 2.6.24. In these tests the full RAM was used to appropriately evaluate caching effects. The system disk was a

Samsung 7200 RPM 250 GB SATA hard drive, while the tests were run on Seagate Cheetah 10k.7 73 GB SCSI hard drives, a Samsung 32 GB Flash SSD (model MCBOE32G8APR-0XA), and a 32 GB Memoright MR25.2-032S. The Samsung drive is a MLC based device and the Memoright drive is a SLC based device. The test hard drives were connected to the system via Adaptec 29320A SCSI controllers. The Samsung and Memoright drives each used adapter devices in order to connect via the SCSI bus. Ext2 was used as the file system on all hard drives. The file systems were remounted between each test to clear file system caches.

A variety of benchmarks were used to test both the sorting and multiple queues approaches to page cache writing. IOzone [IOzone 2006], dbench [Dbench 2008], compilation of the Linux kernel [Linux Kernel 2008], and Postmark were all used to test the system under a variety of situations. Each benchmark was run 5 times and the 95% confidence intervals are shown.

IOzone is a file system benchmark tool with quite a variety of options and settings. IOzone's throughput mode was used with 16 processes. The IOzone benchmark was modified to incorporate a Zipf distribution of write accesses to model some locality. In the experiments, a number of processes accessing files with this Zipf distribution were used to model access locality and some processes accessing files sequentially to represent a mixed workload. The tests were run with 2, 4, and 8 zipfian writers while the remaining 14, 12, and 8 processes performed sequential operations.

Dbench is a benchmark that is designed to emulate the file system load of the netbench benchmark without the requisite networked system set up. It produces the IO

calls that the smdb server in Samba would call when given a netbench run. It runs for a supplied amount of time and during that time attempts to access the file system as fast as possible using its preconfigured load.   Dbench as a benchmark reports application throughput, that being the amount of reads and writes the application completes during its run time. Dbench was run for 300 seconds emulating 100 clients accessing the web server. All the files that were generated were placed onto the test hard drives, and in addition to remounting between each run all of dbench's files were removed before the remount.

In order to test a CPU bound load that still had a large amount of I/O, compilation of the Linux kernel was used. The gcc version that ships with Fedora Core 9 (4.3.0) compiled the Linux kernel source code in the Fedora Core 9 kernel source rpm, kernel-2.6.25-14.fc9.src.rpm. The config file for the kernel was taken directly from the default config file generated for the test system.

Postmark is an I/O intensive benchmark designed to simulate the operation of an e-mail server.   In the Postmark tests, Postmark version 1.5 created 12,500 files between 8 kB and 64 kB in 200 subdirectories and then performed 25,000 transactions. The block size was 4 kilobytes, with the default operation ratios and unbuffered I/O.

In addition to the raw results, a number of different aspects of the data were examined and it was found that different benchmarks improved for different reasons and due to different aspects of the changes made in the system. Among the elements that were examined were the amount of data actually written to the disk, the average size of sequential writes to the disk, and the number of writes within 1 MB boundaries in 100

ms. The first two are fairly self explanatory. The last is designed to examine the boundary crossing effect that Flash drives exhibit. The values were chosen because 1 MB is the usual boundary to suffer a penalty and 100 ms is a reasonable amount of time for the scheduler to use in its calculations.

IOzone

The results for IOzone benchmark are shown in Figures 23, 24, and 25. For each device (Magnetic, Samsung and Memoright), three figures are included corresponding to the three workloads of (2 Zipf writers + 14 sequential), (4 Zipf writers + 12 sequential) and (8 Zipf writers + 8 sequential) for a total of 3x3 = 9 experiments. In each experiment, the performance of the different schemes is normalized to the unmodified base system. Hence, completion times below 1 show performance improvements.
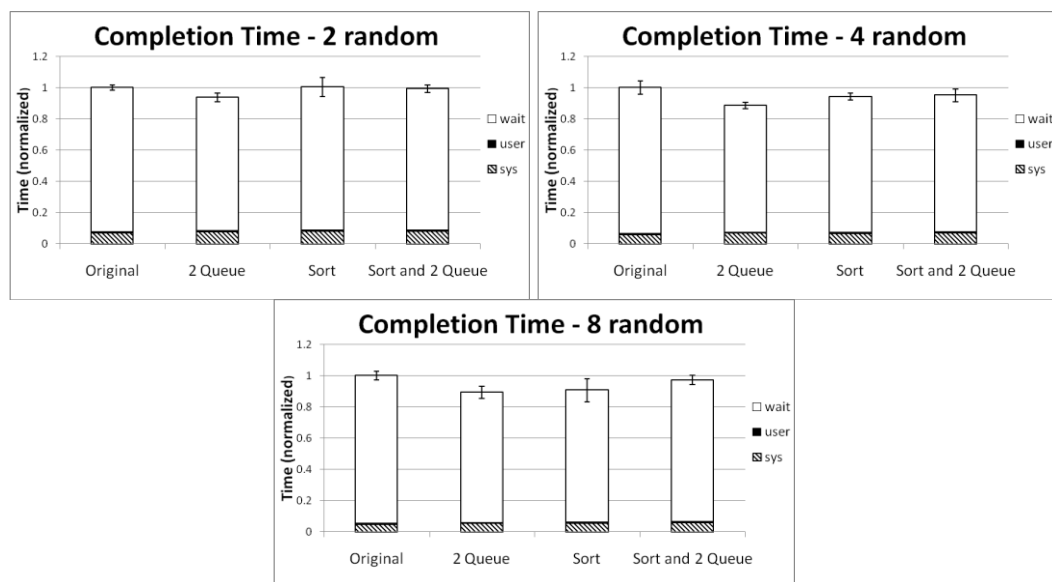
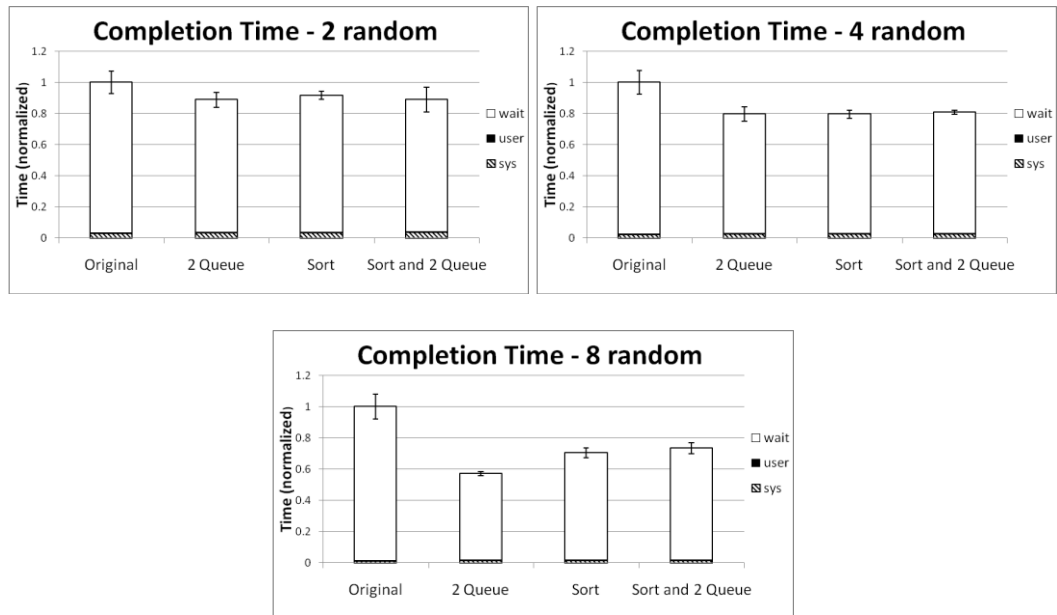Fig. 23.   IOzone magnetic completion times.

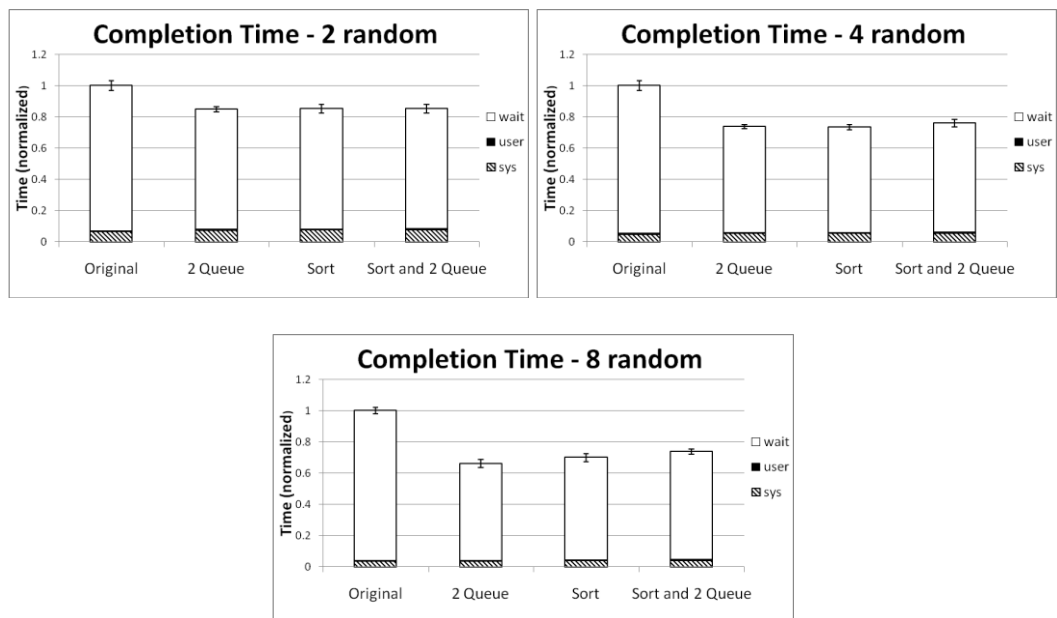Fig. 24. IOzone Samsung completion times.

Fig. 25. IOzone Memoright completion times.

When using the 2 Queue algorithm with 8 zipfian writers the IOzone test showed run times of 89%, 57%, and 66% for the magnetic, Samsung, and Memoright drives respectively. The only run to show an increase of run time was the sorting algorithm on the magnetic drive with 2 Zipfian writers, although even this case was only an increase of .3%. In general, as the workload had a larger portion of zipfian writers, both sorting and the 2 Queue algorithm performed better. As threads which are doing a uniform mix of reads and writes are replaced with writers, improvements to the writing strategy in the cache had a more pronounced effect.

IOzone saw the most improvement among the benchmarks examined. When examining this benchmark's writes within a 1 MB boundary in 100 ms, an obvious pattern occurs. All three modifications, 2 queues, sorting, and 2 queues with sorting shift the writes such that additional writes occur within 1 MB boundaries. The results for the Memoright 8 zipf writers case is shown in Figure 26. Similar, although less dramatic results are seen for 4 and 2 zipf writers. The situations with less than 16 writes in a 1 MB boundary are dramatically reduced while increasing the times that more than 16 writes are performed in 100 ms. This shifting reduces the boundary crossing penalty and thus improves performance.
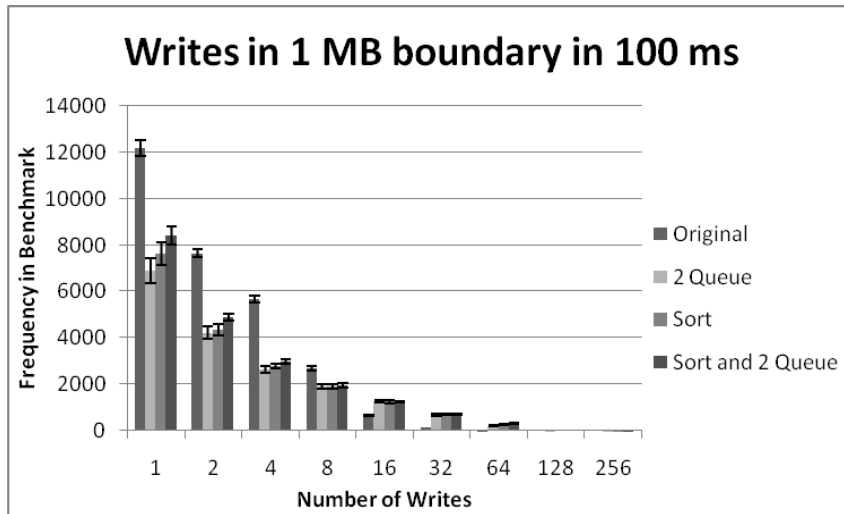
Fig. 26. Number of writes within 1 MB boundary in 100 ms on Memoright drive.

Dbench

Results for the dbench benchmark are shown in Figure 27. For each device (Magnetic, Samsung and Memoright), the application throughput is shown in a separate graph. In each experiment, the performance of the different schemes is normalized to the unmodified base system. Hence, throughput above 1 shows performance improvement.

The dbench tests show uniform improvement with the modified systems. The sorting algorithm provides the best results in this test, with improvements of approximately 19% more throughput for the magnetic drive, 13% more throughput for the Samsung SSD, and almost 33% more throughput for the Memoright SSD. The results are shown in Figure 27. The 2 Queue algorithm showed the least improvement while the Sorting and 2 Queue algorithms combined show improvement in between each algorithm alone.
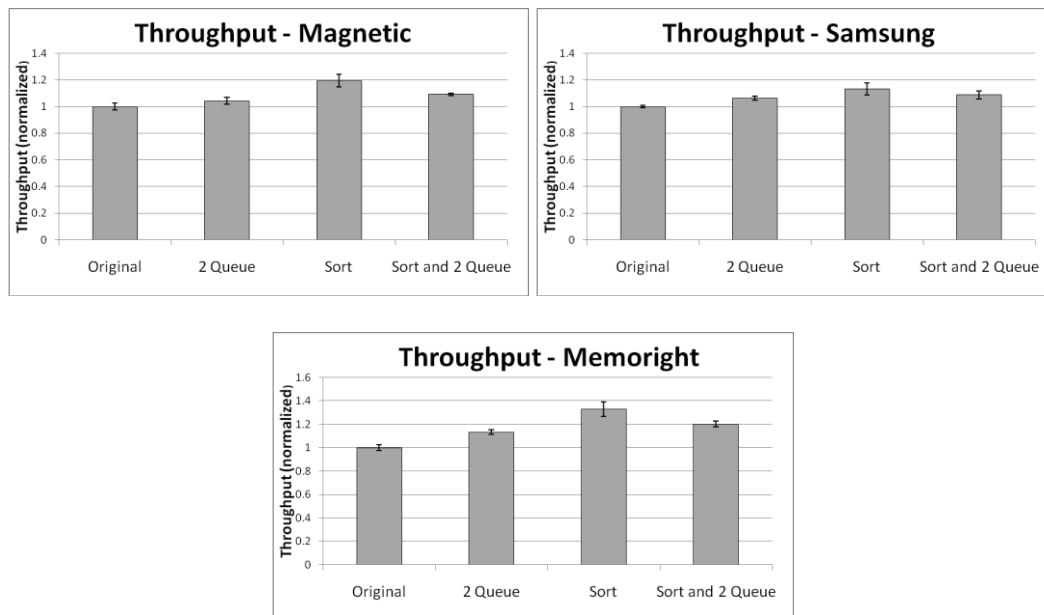
Fig. 27. Dbench throughput.

As can be seen from the results with dbench, both the multiple queues and the sorting algorithms provide improved performance compared to the standard Linux write behavior. We infer that this improvement comes from improvements in the cache hit ratios due to information about the amount of data written to disk. Results are shown for the Memoright drive, as it had the best performance improvement and best shows the relevant issues.

As can be seen in Figure 28, the disk is written with approximately the same amount of data, regardless of the write policy. The read numbers were similarly unchanged, although due to the size of the test, most of the data set stayed in the cache and few reads went to disk. Because the application saw higher throughput while the disk saw the same amount of activity in the same amount of time (300 seconds), less

write data was sent to the disks per transaction by the benchmark in the modified systems. This improvement can be attributed to the modifications to the page cache write-out policies.
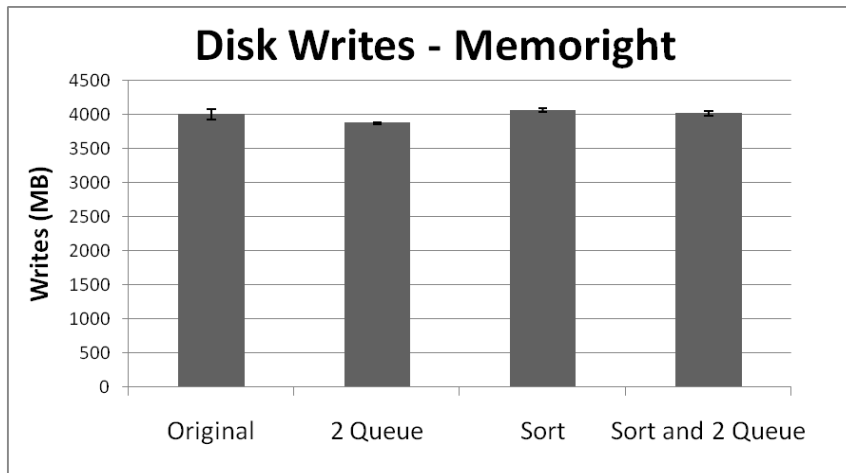

Fig. 28. Data written to disk on Memoright drive during dbench test.

This improvement can be further examined with the data in Figures 29 and 30. Figure 29 shows shows the number of writes to particular blocks on the Memoright drive with the original policies in place, and Figure 30 shows the sorting policy. Blocks that received a single write are omitted for clarity of the graphs. The sorting policy clusters the writes more effectively than the traditional policy, leading to more locality on the disk. Additionally, by sorting the writes, some blocks are rewritten many more times than other blocks. Particularly due to the threshold policy in Linux, if writes can be moved from multiple blocks and focused on a single block, throughput can increase. This is because the multiple blocks are overwritten in the cache repeatedly and do not

increase the dirty ratio of the system. Thus these applications do not pause to wait for writes to finish and can do additional work. Due to the nature of the dbench benchmark, that work is often operations other than writes and so an application can do many operations without waiting on disk I/O as it would be forced to if the system contained more dirty pages.
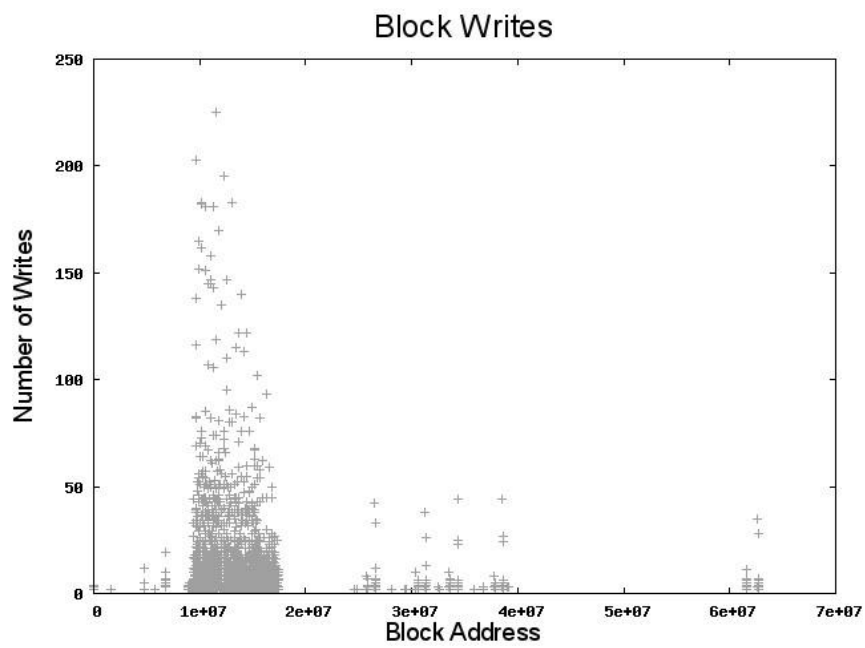

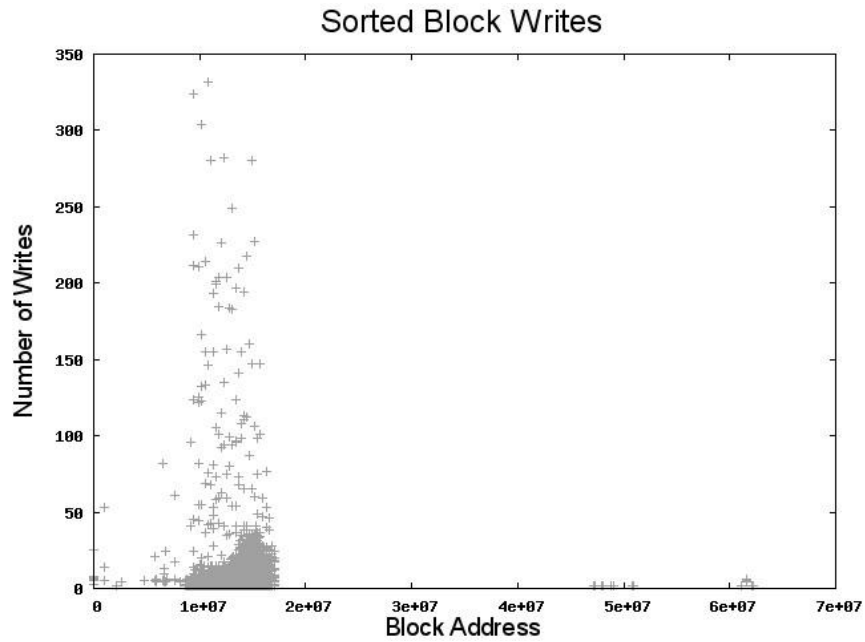Fig. 29 Writes to particular block addresses during dbench test.

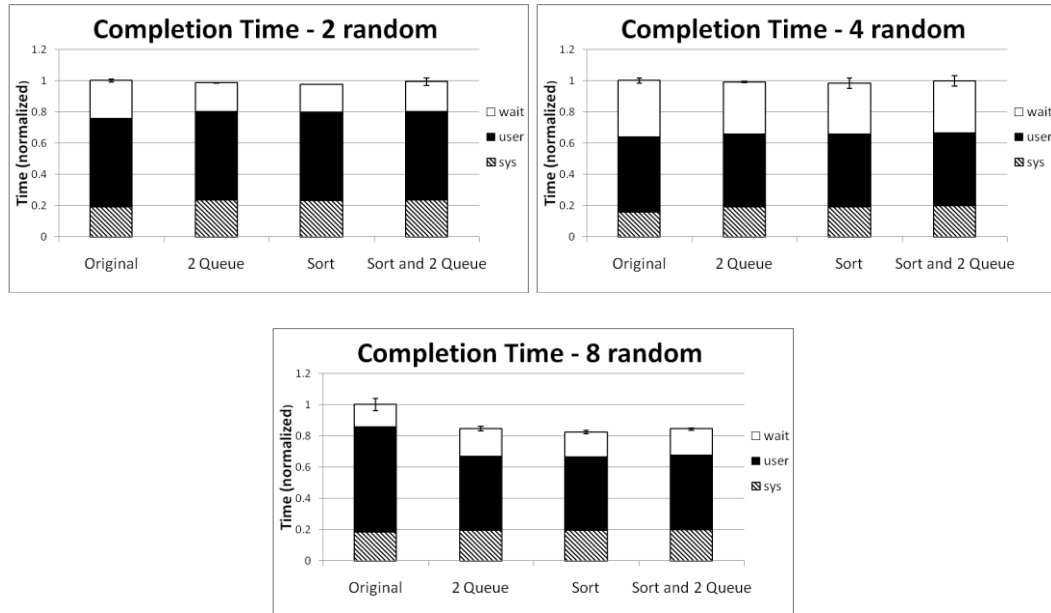Fig. 30. Writes to particular block addresses during dbench test when writes are sorted.

Fig. 31. Linux kernel compilation.

Kernel Compilation

The results from the compilation benchmark are shown in Figure 31. Completion time of the different schemes is normalized to the unmodified base system, so completion times below 1 show performance improvement. Each device is graphed separately.

Compilation shows very little change depending on the page cache algorithm on the magnetic and Samsung drives, although there is a slight improvement. On the Memoright drive, compilation time takes between 82% and 84% of the time when run without any modification to the page cache. Noticeable in all of these tests is an increase in the time spent in kernel mode, although that is accompanied by less time spent waiting. This generally indicates that the algorithms are improving throughput, but most of the time these improvements are somewhat offset by the additional time spent sorting, marking flags, etc. Since this benchmark is processing intensive, the gains achieved through write-out policy modifications are being offset by the extra processing time needed to implement these policies.
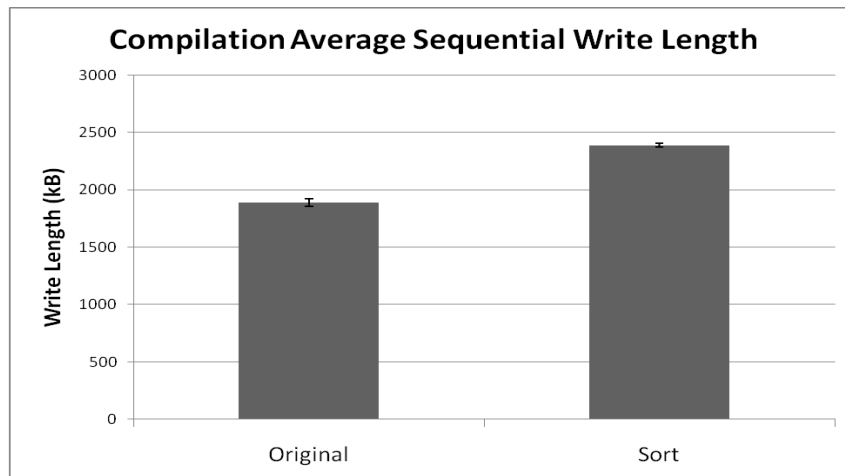
Fig. 32.   Average write length for compilation test.

The memoright drive showed significant improvement with the compilation benchmark. In the case of this benchmark, the average size of sequential writes was also examined. While, in general, roughly the same percentage of writes were sequential when arriving at the disk, the average size of these sequential writes increased by over 26%, as can be seen in Figure 32. The increased length of sequential writes in particular on the compilation benchmark helped to improve the performance.

Postmark

The results of the Postmark benchmark are shown in Figure 33. One graph is shown for each device used in the tests. Completion time of the benchmark is normalized to the unmodified base system, so total times less than 1 show performance improvement.
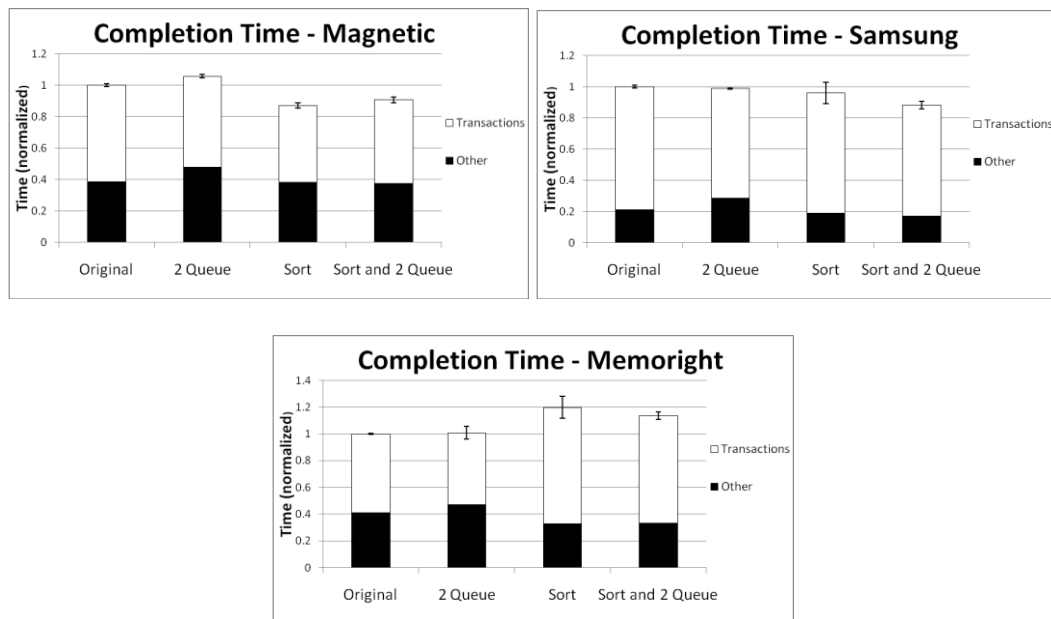
Fig. 33. Postmark completion times.

Postmark shows mixed results in the tests. While showing some improvement on the magnetic disk, postmark showed relatively less improvement with the Samsung SSD, and showed worse performance with the Memoright SSD. As this test focuses on many small files, it is understandable that the overhead might increase relative to the benefit provided by the systems. Particularly, the multiple queues concept is ineffective, as all writes are appends, so a single block will not receive the number of writes necessary to transition from sequential to repeated access.

Different Policies on Different Drives

UmbrellaFS can be configured to allow different cache writeback policies on different underlying file systems and devices. For the tests which provide examples for this ability, IOzone was used again. The 8 sequential and 8 zipf writers case was used with 4 of each type of writer mapped to a drive. Two instances of IOzone were run

simultaneously, each spawning 8 processes which were split 4 and 4 between sequential and zipf. The total time was calculated by adding the times from each IOzone instance together.
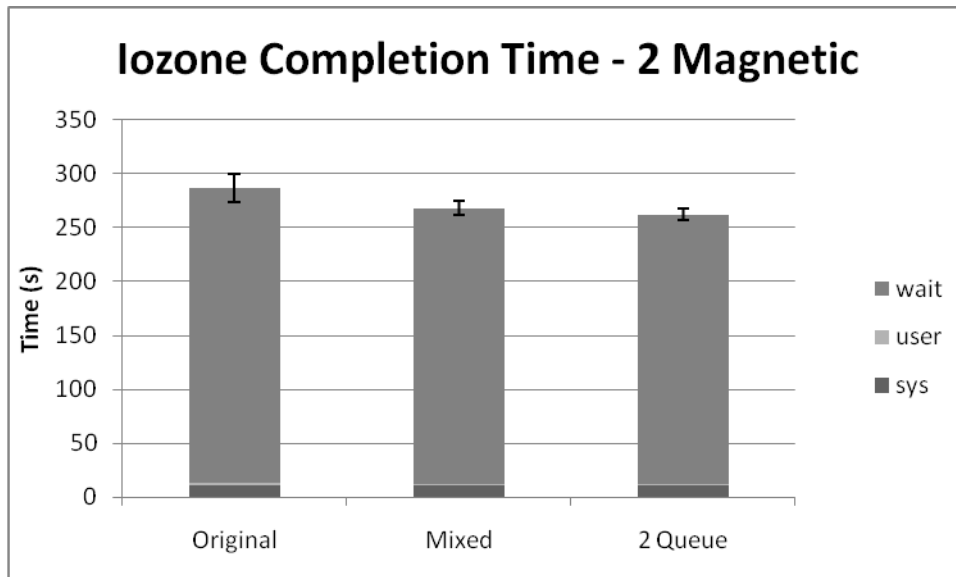

Fig. 34.   IOzone completion time when spread across two magnetic drives.

Figure 34 shows the results of a test where IOzone was run as described above on two magnetic Seagate Cheetah drives. The results show that a 2 Queue implementation on both devices results in 91% of the runtime of the traditional policy, while using 2 Queue on one drive and the traditional policy on the other results in 93% runtime. These numbers are comparable to the 89% runtime difference shown when all 16 processes operated on a single magnetic disk (Figure 23), and show that partial application of the policies presented results in conveying a portion of the benefits of the different policies, as would be expected by only applying the policies to a portion of the disks.
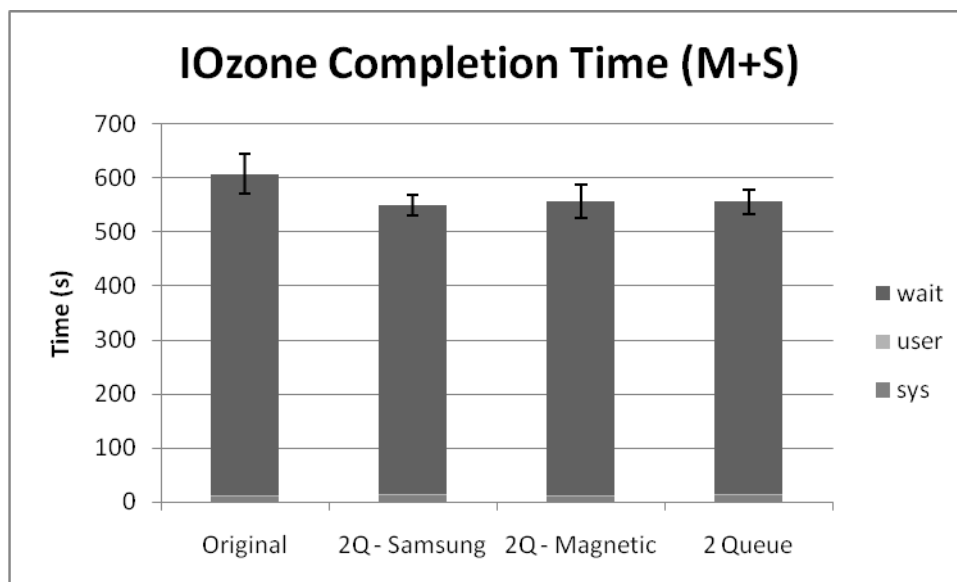
Fig. 35. IOzone completion time spread across one magnetic and one flash drive.

Figure 35 shows the results from running the experiment with one Seagate Cheetah drive and the Samsung Flash drive. In this experiment, in addition to running the test with 2 Queue on both drives and the original policy on two drives, both sets of 2 Queue on one drive and original policy on the other drive were run. Thus the "2Q – Samsung" bar indicates the situation with 2 Queue on the Samsung flash drive and the original policy on the magnetic drive, and "2Q – Magnetic" indicates 2 Queue on the magnetic disk and the original policy on the Samsung flash drive. The results show benchmark completion times of 91% for 2 Queue on only the Samsung device and 92% for both of the other modified policies. Since the Samsung device shows the most benefit in the original IOzone tests, it is unsurprising that implementing 2 Queue on the Samsung device alone is approximately as effective implementing it on both drives. The

benefit is similar to that obtained by only implementing it on the magnetic disk, which is a bit surprising.

NILFS

In addition to running the various tests on a traditional ext2 file system, tests were run with NILFS as the file system. NILFS is a log-structured file system under active development for Linux. The results for IOzone are shown in Figures 36, 37, and 38. The results for dbench, Postmark, and compilation showed similar results.
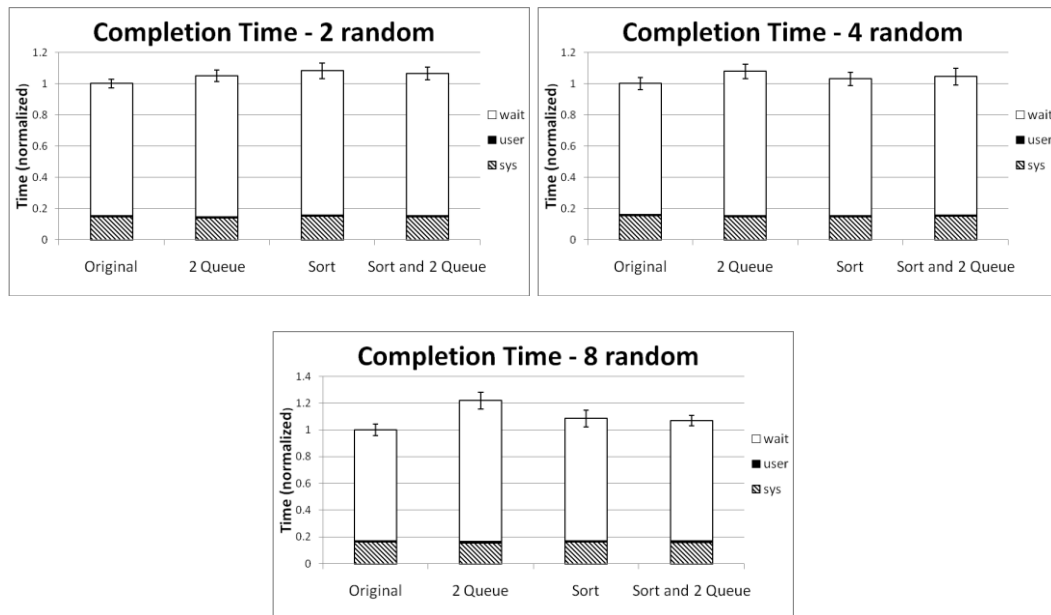

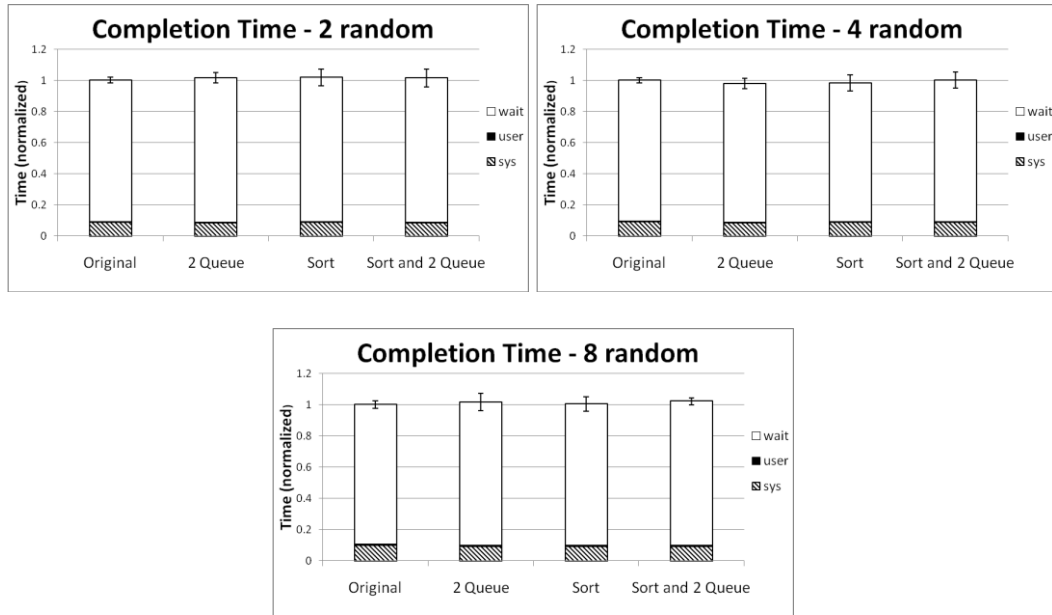Fig. 36. IOzone magnetic completion times on NILFS.

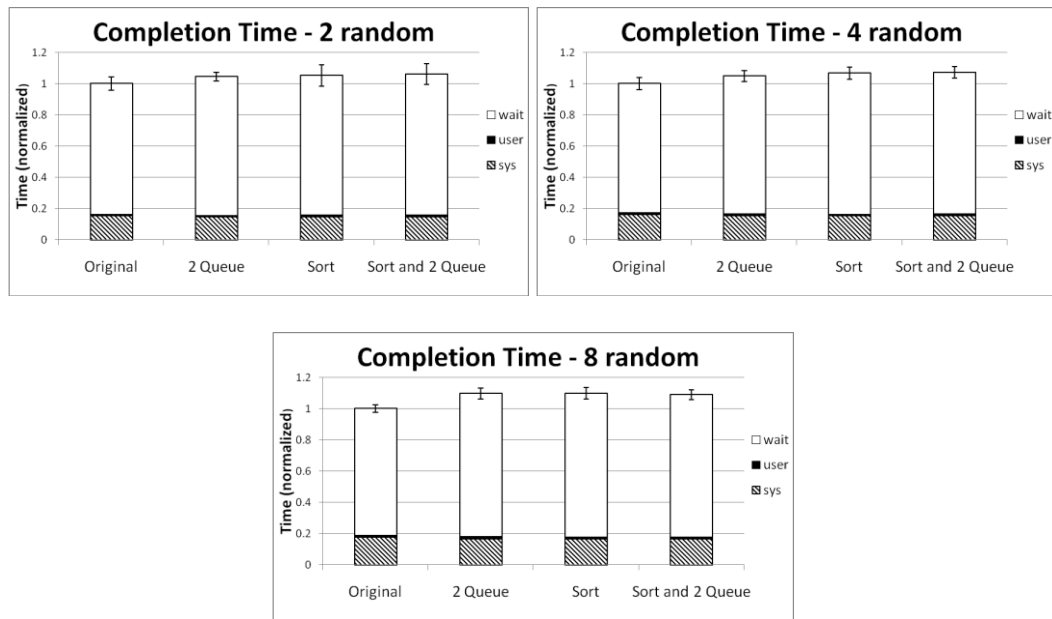Fig. 37. IOzone Samsung completion times on NILFS.

Fig. 38. IOzone Memoright completion times on NILFS.

The results for tests on NILFS do not show significant improvement, and in most cases show a slight penalty to using the various sorting and 2 Queue strategies. The tests do not show improvement when combined with NILFS in large part because NILFS converts these benchmarks into sequential write operations. With an unmodified page cache, the benchmarks simply run as fast as the device can write. Reordering the writes coming out of the page cache cannot improve on the device's maximum write speed, and the reordering imposes some overhead. Thus the system runs at approximately the same speed with only the addition of a bit of overhead in the page cache.

Trace Results

The dbench benchmark was modified slightly in order to allow the playing of trace files. The trace files were obtained from the Storage Networking Industry Association (SNIA) trace repository [SNIA 2009], and the result shown is from an NFS trace of email and workloads from Harvard's division of engineering and applied sciences. Because the NFS traces are in a format that dbench cannot run, they had to be converted before the actual test. Additional work was also done to prepare the system to run the trace. During the test run, the trace was replayed at the maximum speed the system could handle rather than the original speed of the trace.

In order to convert the trace into a file that dbench could run, the trace was parsed to convert the commands in the trace to the format that dbench uses. In cases where dbench does not have an analogous function to the NFS trace, that operation was simply omitted. In addition to this conversion of the trace, it is necessary to prepare the file system to run the trace. Because the NFS traces did not start when the file system

was created, the traces would call operations on files that were not created in the trace. For example, a file might be read in the trace, but never created or any data written to it. If this operation were to be attempted without a file present, the read would fail. Since the read did not fail in the trace itself, this would represent a departure from the trace to the replay. Issues such as this were resolved by tracking files throughout the trace. At the end of the conversion to dbench's format, and additional dbench formatted file was created. This file was designed to create a file system image that would be able to successfully handle the operations in the trace. Files and directories were created in a manner consistent with the first appearance of a file or directory in the trace. Files were filled with data to satisfy the initial size of the file detected in the trace. While this does not necessarily create a file system image that is a duplicate of that in the original trace, it does create a file system image that is close enough to run the trace itself.

Figure 39 shows the results of the replaying of one hour of the traces obtained from SNIA. In each case the completion time is only shown for the trace run, not the initialization necessary to prepare the system. The results of the trace replaying show overhead imposed by all but the combination of sorting and 2 Queue on the Samsung drive. That result is very close to the unmodified system and well within the 95% confidence interval for the unmodified page cache.

While this representation of trace replay does not necessarily exactly mimic the situation from the traces themselves, it does indicate that the proposed changes to the page cache system are not necessarily beneficial in all situations.
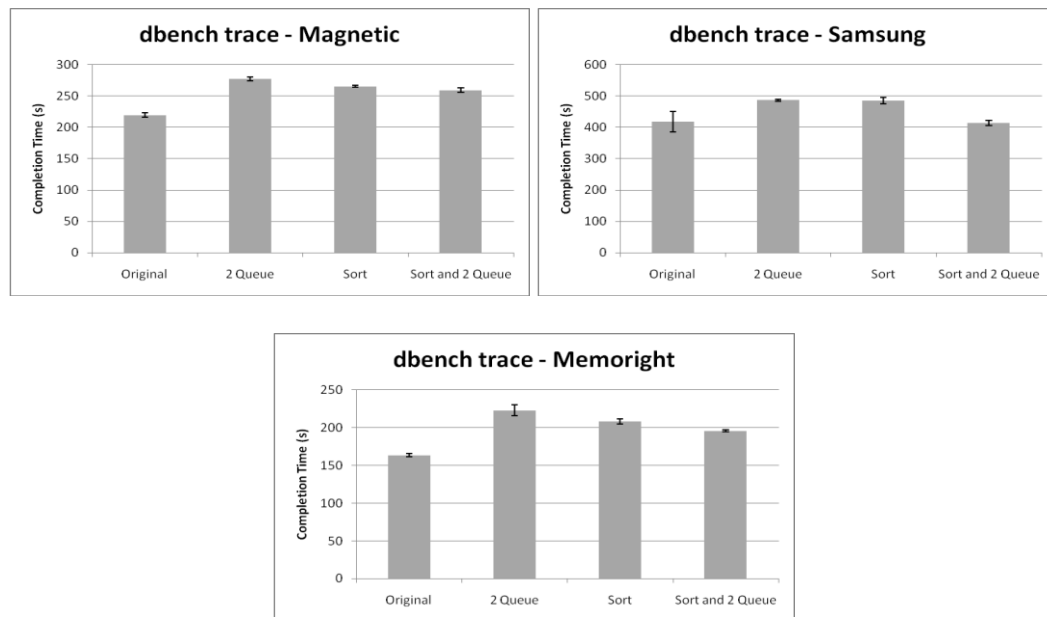
Fig. 39. Dbench trace completion times.

The dbench trace runs performance with respect to sorting is not unexpected, considering the nature of how the file system is set up. Because the initialization process is done based on the order of the files being referenced, the files are effectively referenced in a sorted order on disk. Therefore the sorting itself results in very little change in the order of writes with an increase of overhead.

With respect to the 2 Queue strategy, the nature of the trace is an issue. The 2 Queue strategy relies upon the workload having both sequential and non-sequential operations. In the traces from SNIA, over 90% of the files are typically written sequentially. Writes to individual pages are also classified as sequential by the algorithms over 90% of the time. With less than 10% of the files classified as non-sequential, the 2 Queue strategy does not have enough difference in write patterns to work with to generate a benefit.

Optimality

The proposed cache strategies were evaluated with respect to optimal possibilities in order to ascertain the value of improvement provided. Due to the nature of the benchmarks, the trace results were the only ones that could be evaluated for optimality. The other benchmarks lack elements such as future knowledge that are necessary to evaluate the results from an optimal standpoint.

For the purposes of this section, optimality is examined from a number of angles. In the first, a clairvoyant algorithm was used to determine cache eviction policies. This is the traditional "optimal" caching strategy, and in this situation writes were not performed until a dirty page was selected to be evicted from the page cache. In addition to an unconstrained optimal algorithm, efforts were made to simulate optimization with respect to amount of data written to disk.

For the data write minimizing simulation, a moving 30 second window was used to emulate the 30 second rule. Files were held in the cache until the 30 second maximum to provide the most opportunity to have repeated writes to the page cache and save writes to disk. Using constraints such as the 30 second rule provided a better understanding of what the best possible situation could be in a live system with reasonable constraints. In addition, the write minimizing did not require future knowledge as the optimal replacement algorithm does.

All simulations assumed a 2 GB cache. Results for both simulations are evaluated from the standpoint of boundary crossings and total data written to disk, since

evaluation based on completion time would depend on the particulars of the underlying storage device.

Boundary Crossings

Figure 40 shows the number of boundary crossings for a variety of policies, including the "optimal" strategy and measured results from the previous trace runs. The results from the total write minimizing strategy are not included for the sake of clarity. The write minimizing strategy that is not shown in this graph did not take into account efforts to minimize boundary crossing, and lead to a dramatic increase in boundary crossings that made representation with optimal, the original, and Sort + 2 Queue strategies difficult.
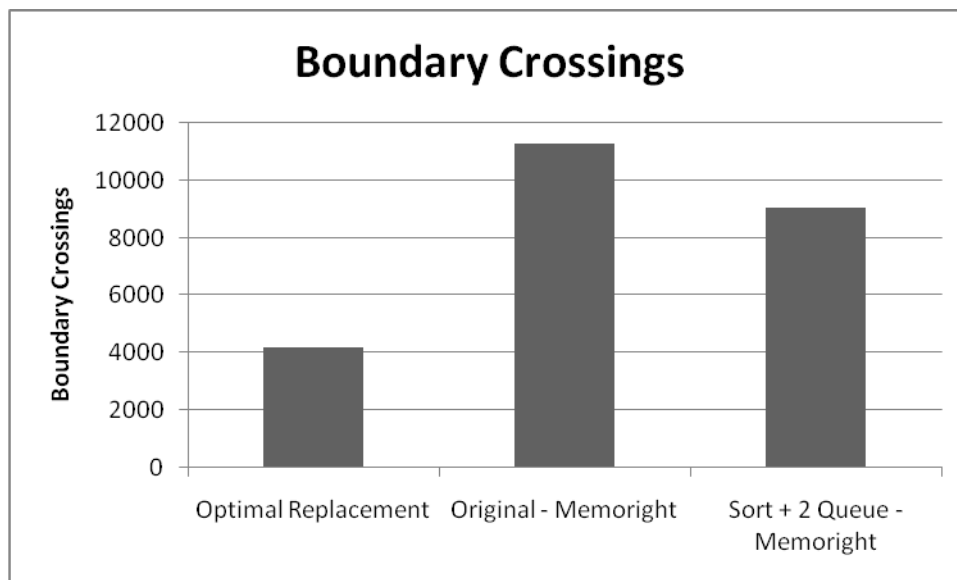

Fig. 40. Boundary crossings.

From Figure 40 we see that the optimal strategy reduces the boundary crossing penalties dramatically. The values for the original page cache writeback strategy and the combined sorting and 2 Queue strategies are measured from block level tracing of the trace replay. These show that the strategies presented make improvements in the total number of boundary crossings. Obtaining results comparable to the optimal replacement strategy would be difficult, given elements such as future knowledge that that strategy uses in its decision making. In addition, the optimal replacement strategy does not consider real system limitations such as the 30 second constraint on length a page can remain dirty. It does consider the size of the cache, however.

Amount of Writes

Figure 41 shows the amount of data written for the optimal replacement algorithm, the write optimized (and 30 second rule constrained) algorithm, and measured results from trace replay on the Samsung drive. These results show that there is relatively little room for improvement with respect to total writes when the 30 second rule is still used. Without the 30 second rule, a dramatic reduction is possible, as seen by the results for the optimal replacement algorithm. The modified strategies presented in this dissertation do not dramatically affect the total number of writes, as is demonstrated by the measured results from the Samsung drive.
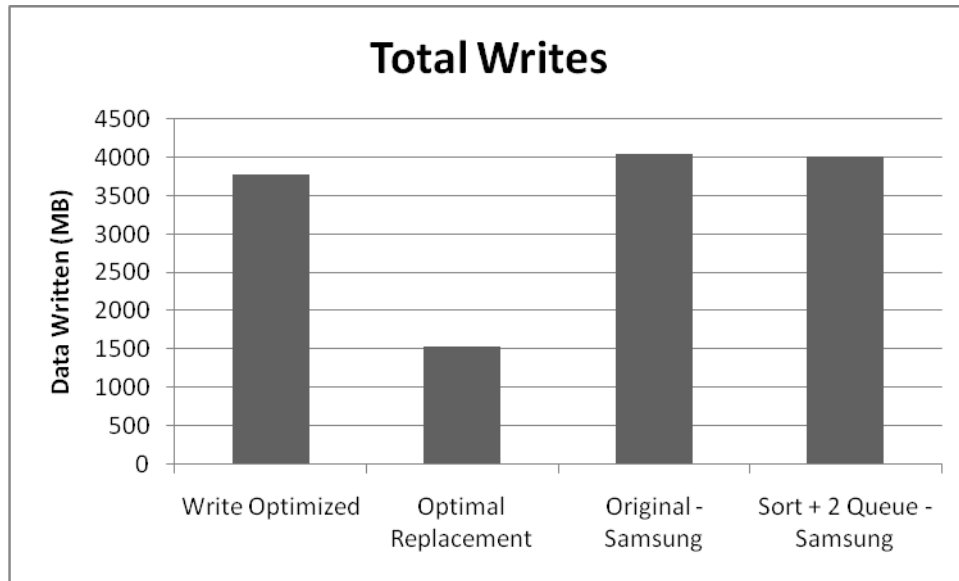
Fig. 41. Total amount of data written.

CHAPTER VI

CONCLUSION

Conclusion

The Umbrella File System provides myriad opportunities for system level improvement with respect to storage solutions. UmbrellaFS provides opportunities to exploit device level differences, such as those presented by the advent of Flash SSDs, without requiring particular action by end users. The prototype implementation of UmbrellaFS on a 2.6 Linux kernel presented in this dissertation adds little overhead to most file system operations. In addition, the prototype has shown the ability of UmbrellaFS to work in a live system consisting of both Flash SSDs and RAID arrays and to work with the selective encryption of files. In addition to these examples, UmbrellaFS has provided the ability to apply different cache writeback policies in the page cache.

Changing the write out policy of the page cache can achieve significant improvements in a variety of workloads that have mixed random and sequential access characteristics. Both using multiple queues to distinguish between various different types of access patterns and sorting the writes leaving the page cache provided improvements in performance through improvements in increased sequentiality of writes, and through reducing boundary crossing penalties in SSDs. These improvements were found to be effective in improving performance of both traditional magnetic disks and Flash SSDs.

While clearly not a panacea for page cache writing, the techniques show improvements of over 40% in some circumstances, and they do not negatively impact the system in most situations. Most importantly, the use of UmbrellaFS to target the techniques to particular file systems while not affecting other parts of the system presents an opportunity to tailor not only applications and file systems to appropriate devices, but to tailor particular page cache techniques to file systems and devices. This represents a new dynamic that has the potential to provide new capabilities and better performance in storage systems.

Future Work

Providing hysteresis to the UmbrellaFS system could help to minimize the negative effects of oscillation. The ability to include file system based rules, such as rules concerning the current available space in file systems or other aspects that are not necessarily present in the file metadata, would provide additional ways to control file placement. Inferring beneficial policy rules from file access behavior in guiding the policy decisions is another area where UmbrellaFS could be expanded.

In addition to enabling the selective application of cache writeback policies, UmbrellaFS could be expanded to provide selective operations at additional layers of the storage stack. Areas such as the device drivers and the Virtual File System will be explored to determine where the additional abilities of UmbrellaFS might be able to influence decisions and in what manner that influence might provide the best results.

A more efficient sorting algorithm will be incorporated in the future. By using a sorting algorithm that operates in $O(n \log n)$ time as opposed to the current $O(n^2)$

selection sort the overhead can be reduced particularly in cases with a very large number of small files in the system.

Mixed workloads of reads and writes are particularly difficult for Flash drives, and further efforts will be devoted to improving performance in these cases. Additional classification mechanisms and categories provide opportunities for improved performance.

Efforts will be made to adapt write thresholds in a manner similar to AWOL without combining page cache the page cache into other separate elements of the kernel. In particular, it would be preferable to do this without combining the memory manager and the I/O scheduler, and this will be a focus of future efforts.

Finally, known sequential detection and mechanisms such as balancing recency and frequency information in decision making will be incorporated into the page cache writeback mechanisms.

REFERENCES

ANDERSON, D. C., CHASE, J. S., AND VAHDAT, A. M. 2000. Interposed request routing for scalable network storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI).* San Diego, CA. 259-272.

BAEK, S., AHN, S., CHOI, J., LEE, D., AND NOH, S. 2007. Uniformity improving page allocation for flash memory file systems. In *Proceedings of ACM EMSOFT*. Salzburg, Austria. 154-163.

BATSAKIS, A., BURNS, R., KANEVSKY, A., LENTINI, J., AND TALPEY, T. 2008. AWOL: An adaptive write optimizations layer. In *Proceedings of FAST '08*. San Jose, CA. Article 5.

BHARATHI, S., KIM, B.K., CHERVENAK, A., AND SCHULER, R. 2005. Combining virtual organization and local policies for automated configuration of grid services. In *Proceedings of Self-Organization and Autonomous Systems in Computing and Communications (SOAS) 2005*. Glasgow, UK. 194-202.

BLAICH, A., LIAO, Q., ALLAN, G., STRIEGEL, A. AND THAIN, D. 2007. Lockdown: Distributed policy analysis and enforcement within enterprise network. 16th Usenix Security Symposium, poster. Boston, MA.

CHANG, L. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*. Seoul, Korea. 1126-1130.

CHEN, F., JIANG, S., AND ZHANG, X. 2005. Clock-pro: An effective improvement of the clock replacement. In *Proceedings of USENIX '05*. Anaheim, CA. 35-49.

CHEN, F., KOUFATY, D., AND ZHANG, X. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of SIGMETRICS 2009*. Seattle, WA. 181-192.

COKER, R. 2001. Bonnie++. http://www.coker.com.au/bonnie++/. Accessed Jan. 20, 2009.

DBENCH. http://freshmeat.net/projects/dbench/. Accessed August 20, 2009.

DUNN, M., AND REDDY, A. L. N. 2009. A new I/O scheduler for solid state devices. Texas A&M University ECE Technical Report TAMU-ECE-2009-02.

FORNEY, B., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2002. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proceedings of FAST '02*. Monterey, CA. Article 5.

FRØLUND, S., MERCHANT, A., SAITO, Y., SPENCE, S., AND VEITCH, A. 2004. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Boston, MA. 48-58.

GAL, E. AND TOLEDO, S. 2005. Algorithms and data structures for flash memories. *ACM Computing Surveys, 37*, 2, 138-163.

GANGER, G. R., STRUNK, J. D., AND KLOSTERMAN, A. J. 2003. Self-* Storage: Brick-based storage with automated administration. Carnegie Mellon University Technical Report CMU-CS-03-178.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY. 29-43.

GILL, B., KO, M., DEBNATH, B., AND BELLUOMINI, W. 2009. STOW: A spatially and temporally optimized write caching algorithm. In *Proceedings of USENIX 2009.* San Diego, CA. 327-340.

GILL, B. S., AND MODHA, D. 2005. WOW: Wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of FAST 2005.* San Francisco, CA. 129-142.

GOUGH, V. 2006. EncFS encrypted file system. http://www.arg0.net/encfs. Accessed Feb. 11, 2010.

HALCROW, M. 2005. eCryptfs: An enterprise-class cryptographic file system for Linux. In *Proceedings of the Linux Symposium.* Ottawa, Ontario, Canada. 1:201-218.

HOHMANN, C. 2007. CryptoFS. http://reboot.animeirc.de/cryptofs/. Accessed August 14, 2007.

IOZONE FILESYSTEM BENCHMARK. http://www.iozone.org/. Accessed Feb. 11, 2010.

JIANG, S., DING, X., CHEN, F., TAN, E. AND ZHANG, X. 2005. DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of FAST 2005*. San Francisco, CA. Article 8.

JIANG, S., AND ZHANG, X.. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review, 30*, 1, 31-42.

JOHNSON, T., AND SHASHA, D. 1994. 2Q: A low overhead high performance buffer management replacement algorithm, In *Proceedings. of VLDB '94.* Santiago de Chile, Chile. 439-450.

JOUKOV, N., KRISHNAKUMAR, A. M., PATTI, C., RAI, A., SATNUR, S., TRAEGER, A., AND ZADOK, E. 2007. RAIF: Redundant Array of Independent Filesystems. In *Proceedings of the 24th IEEE Symposium on Massive Storage Systems and Technologies*. 199-214.

KATCHER, J. 1997. Postmark: A new file system benchmark. Network Appliance Technical Report TR3022. http://www.netapp.com/tech library/3022.html. Accessed August 30, 2004.

KIM, H., AND AHN, S. 2008. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of FAST 2008*. San Jose, CA. Article 16.

KIM, H., AND LEE, S. 2002. An effective flash memory manager for reliable flash memory space management. *IEICE Trans. on Information Systems E85-D,* 6, 950-967.

KISTLER, J., AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems, 10*, 1 (February 1992), 3-25.

KLEIMAN, S. R. 1986. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association: Summer Conference Proceedings*. Atlanta, GA. 238-247.

LEE, D., CHOI, J., KIM, J. H., NOH, S. H., MIN, S.L., CHO, Y., AND KIM, C. S. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers, 50*, 12, 1352–1360.

LEE, H., AND BAHN, H., 2009. Characterizing virtual memory write references for efficient page replacement in NAND flash memory. In *Proceedings of MASCOTS 2009.* London, UK, Article 44.

LEE, J., KIM, S., KWON, H., HYUN, C., AHN, S., CHOI, J., LEE, D., AND NOH, S. 2007. Block recycling schemes and their cost-based optimization in nand flash memory based storage system. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software.* Salzburg, Austria. 174-182.

LINUX KERNEL. 2008. Linux 2.6.25 kernel source code. http://www.kernel.org. Accessed Feb. 11, 2010.

MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. 2006. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor System.* Boulder, CO. 195-208.

MEGIDDO, N. AND MODHA, D. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of FAST 2003.* San Francisco, CA. 115-130.

O'NEIL, E.J., O'NEIL, P.E., AND WEIKUM, G. 1993. The LRU-K page replacement algorithm for database disk buffering, In *ACM SIGMOD Record, 22*, 2, 297-306.

OPENBSD. 2007. OpenSSH. http://www.openssh.org. Accessed April 11, 2007.

PANASAS, INC. 2005. Object storage architecture white paper. http://www.panasas.com/docs/Object_Storage_Architecture_WP.pdf. Accessed Feb. 11, 2010.

PAPATHANASIOU, A. E., AND SCOTT, M. L. 2004. Energy efficient prefetching and caching. In *Proceedings of USENIX'04*. Boston, MA. 255-268.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data.* Chicago, IL. 109-116.

PATTERSON, R. H., GIBSON, G. A., AND SATYANARAYANAN, M. 1992. Using transparent informed prefetching (TIP) to reduce file read latency. In *Proceedings of the NASA Goddard Conference on Mass Storage Systems.* Greenbelt, MD. 329-342.

PROJECT T10. 2004. Information technology—SCSI object-based storage commands (OSD), Rev. 10. http://www.t10.org. Accessed Feb. 11, 2010.

SAMSUNG SEMICONDUCTOR EUROPE. 2007. Samsung flash solid-state drive. http://www.samsung.com/eu/Products/Semiconductor/downloads/Samsung_SSD_for _mobile.pdf. Accessed December 3, 2007.

SEAGATE TECHNOLOGY LLC. 2007. Seagate unveils new giants -- 250GB notebook hard drive and the first encrypting 1TB desktop PC drive. http://www.seagate.com. Accessed Feb. 11, 2010.

SHENOY, P., GOYAL, P., AND VIN, H. 1998. Architectural considerations for next generation file systems. University of Massachusetts Technical Report UM-CS-1998-048.

SNIA IOTTA Repository, 2009. http://iotta.snia.org. Accessed Feb. 11, 2010.

SUN MICROSYSTEMS. 2004. ZFS: The last word in file systems. http://www.sun.com/2004-0914/feature/index.html. Accessed Jan. 19, 2005.

SYMANTEC. 2007. HP and Symantec: Enabling ILM solutions with dynamic storage tiering. Symantec White Paper, http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_hp_symantec_dynamic_storage_tiering.pdf. Accessed Feb. 11, 2010.

SZEREDI, M. 2007. Filesystem in Userspace. http://fuse.sourceforge.net/. Accessed Feb. 11, 2010.

TRAEGER, A., ZADOK, E., JOUKOV, N., AND WRIGHT, C. P. 2008. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS), 4*, 2 (May 2008), 1-56.

TRAINOR, M. 2006. Overcoming disk drive access bottlenecks with Intel® Robson technology. *Technology@Intel Magazine, 4*, 9, 9-11.

VAHDAT, A., DAHLIN, M., ANDERSON, T., AND AGGARWAL, A. 1999. Active names: Flexible location and transport of wide-area resources. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS),* Boulder, CO. 151-164.

WANG, A., KUENNING, G., REIHER, P., AND POPEK, G. 2006. The Conquest file system: better performance through a disk/persistent-RAM hybrid design. In *ACM Transactions on Storage (TOS), 2*, 3, 309-348.

WANG, J. AND HU, Y. 2002. WOLF – A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems. In *Proceedings of FAST 2002*. Monterey, CA. 47-60.

WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. 1996. The HP AutoRAID hierarchical storage system. In *ACM Transactions on Computer Systems, 14*, 1, 108-136.

WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., ZADOK, E., AND ZUBAIR, M. H. 2004. Versatility and Unix semantics in a fan-out unification file system. Stonybrook University Technical Report FSL-04-01b.

Wu, X., and Reddy, A. L. N. 2009. Managing Storage Space in a Flash and Disk Hybrid Storage System. In *IEEE MASCOTS Conference*. London, UK. Poster.

YOSHIJI, A., KONISHI, R., SATO, K., HIFUMI, H., TAMURA, Y., KIHARA, S., MORIAI, S. 2008. New Implementation of a Log-structured File System, version 2.0.6. http://www.nilfs.org/en/index.html. Accessed Feb. 11, 2010.

ZADOK, E., IYER, R., JOUKOV, N., SIVATHANU, G., AND WRIGHT, C. P. 2006. On incremental file system development. In *ACM Transactions on Storage (TOS), 2*, 2, 1-33.

ZHU, Q., DAVID, F. M., DEVERAJ, C. F., LI, Z., ZHOU, Y., AND CAO, P. 2004. Reducing energy consumption of disk storage using power-aware cache management. In *Proceedings of HPCA'04.* Madrid, Spain. 118-129.

ZHU, Q., SHANKAR, A., AND ZHOU, Y. 2004b. PB-LRU: A self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *Proceedings of ICS'04*. Malo, France. 79-88.

VITA

John Allen Garrison received his Bachelor of Science degree in computer engineering from the University of Tennessee, Knoxville in 2004. He entered the Ph.D. program in computer engineering at Texas A&M University in August 2004. His research interests include storage systems, flash solid state devices, and computer security. His graduate studies were funded by the Department of Defense and plans employment with DoD upon graduation in May 2010.

Dr. Garrison may be reached at: 214 Zachry Engineering Center, TAMU 3128, College Station, TX 77843-3128. His e-mail address is: jgarrison@tamu.edu.