Umbrella File System:
Storage Management Across Heterogeneous Devices

John A. Garrison and A. L. Narasimha Reddy
Department of Electrical and Computer Engineering
Texas A&M University
College Station, TX 77843-3259, USA
{jgarrison, reddy}@ece.tamu.edu

***Abstract*—In this paper, we describe the Umbrella File System (UmbrellaFS), a stackable file system that presents a single namespace to the user while allowing the option for different user files to be placed on different underlying devices. UmbrellaFS is designed to provide flexibility in matching diversity of file access characteristics to diversity of device characteristics through a user or system administrator specified policy. We present the design and results from a prototype implementation of UmbrellaFS on Linux 2.6. The results show that UmbrellaFS has little overhead for most file system operations while providing an ability to map files to devices beyond the one-to-one mapping provided from namespace to devices in current systems.**

Keywords-File systems; Device characteristics; Flash drives; Policy-driven storage; Namespaces.

1. 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. It is necessary for storage systems and file systems to be designed to exploit the unique characteristics of these different devices. While current storage systems exploit devices of different characteristics, some of the characteristics of flash drives such as limited write cycles warrant us to revisit this problem.

With the emergence of solid state flash RAM drives, the differences between different types of devices are becoming more pronounced in some characteristics. Flash drive capacities are approaching the 100 GB range, and their read and write speeds are approaching those of more traditional magnetic disks [18]. These drives have much faster seek times than their magnetic counterparts, due to their lack of moving parts, but often have a limited number of times that the drive can be written. Flash drives also typically have more uniform performance depending on file size, where magnetic disks typically perform better with larger file sizes, as can be seen in Section 4.

In addition to different physical device characteristics, there are many different methods for organizing the storage devices. Various RAID [16] levels have been developed. RAID5, for example, can provide users with high throughput in their storage system, but with a cost to small writes. Mirroring, on the other hand, avoids the small-write penalty of RAID5 while still providing additional reliability, but at the cost of lower throughput. Different storage methods can have different characteristics.

Various methods have been proposed to compensate for and exploit this diversity in device characteristics. Most storage systems use memory for caching and manage memory and storage devices together. Systems such as HP's AUTORAID [26] have been designed to work at the device level by automatically moving blocks between different devices. Flash memory is being used as a cache to improve disk drive performance [e.g., 24]. These methods work at the device level, and thus 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.

While this transparency of device characteristics may reduce the challenges in dealing with devices of different characteristics, it also prohibits flexibility in exploiting these differences to the advantage of individual applications. For example, if a user deems it ideal to store many small read-only files on a flash drive while storing the remaining files on a magnetic disk in a storage system, currently the file systems and storage systems do not provide the user any mechanisms to accommodate such a system organization. While Object Storage Devices (OSDs) [27] can potentially provide such flexibility, OSDs require extensive changes to storage systems, file systems and clients.

Like devices, applications also have remarkable diversity. 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.

In this paper, we propose 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. We propose UmbrellaFS to make this "policy-based storage" possible. We describe the details of UmbrellaFS and our implementation of policy-based storage below.

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. In addition, the file system is in possession of copious amounts of information about the files themselves.

High level hints have been shown to be valuable in improving I/O performance [17]. The file system has access to information which makes it the optimal nexus to leverage application and device differences to improve system performance. 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.

As an example configuration that can benefit from UmbrellaFS, 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 [6] [7] [12] and even device level [19] encryption options available to users. Consider a situation where some user files need to be encrypted on such a system. If we employ an encryption file system encompassing all the devices, we don't harness the hardware encryption support available at one of the devices (or encrypt some files 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 we employ normal file systems on all the drives and rely on the device-level encryption, the user has to carefully place the sensitive files on this device to ensure their protection.

UmbrellaFS can be employed in this scenario to direct files to the appropriate underlying storage device while allowing the files to be viewed in the same directory. This avoids 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).

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, such diversity in devices can be easily mapped to the applications through UmbrellaFS developed in this paper.

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, we give users the option of setting up the mapping from files to underlying storage in a manner that optimizes file usage.

In Section 2 we outline our approach to this problem, and how UmbrellaFS is set up. Section 3 details our prototype, and Section 4 presents benchmark results. A number of example scenarios that show the utility of UmbrellaFS are shown in Section 5. Section 6 presents related work and Section 7 concludes and presents directions for future research.

2. OUR APPROACH

2.1 Hourglass Model

The Virtual File System (VFS) [10] maintains a one to one mapping between the namespace and the underlying file systems. 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 1. This one to one mapping 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.

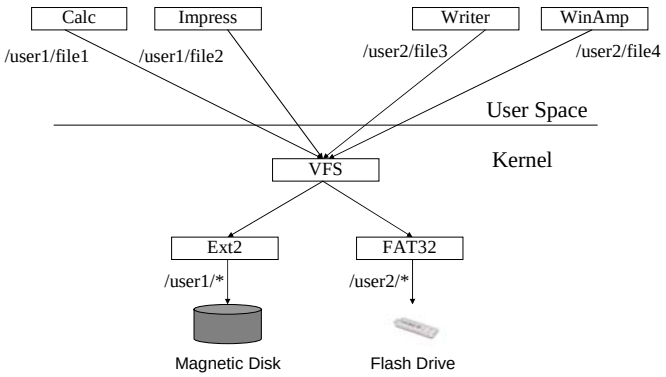

Figure 1. Hourglass Model with traditional VFS

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 namespace. 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, which we call UmbrellaFS, provides the function of unifying the namespaces and demultiplexing the namespace operations among the different underlying file systems.

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 2. 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 2.

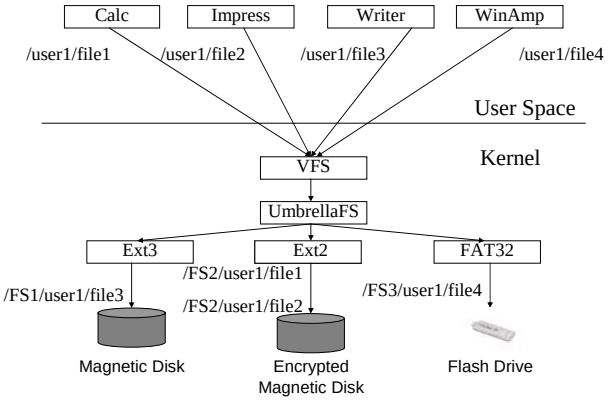

Figure 2. Hourglass Model with UmbrellaFS

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 1, 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 [11]. Based on user-controlled policies, files can be mapped to the underlying device by mapping individual files (/user1/file1, /user1/file2… in Figure 2) to the native file systems (/FS1, /FS2… in Figure 2).

As an example of UmbrellaFS's utility, take 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 3 shows the user's view of the namespace, as well as how the files are stored in the underlying storage.

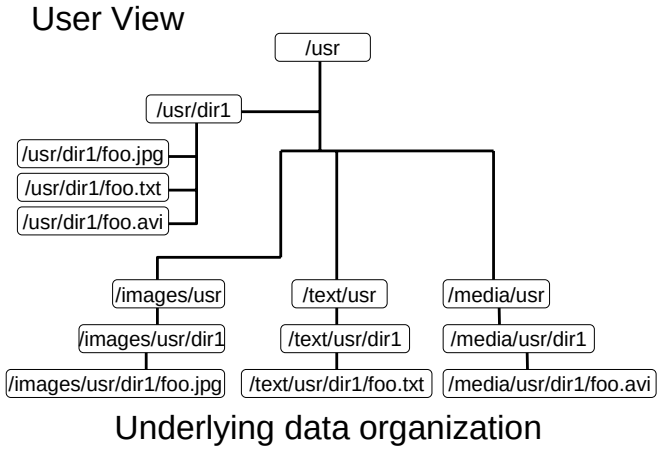

Figure 3. Example mapping of user namespace based on file contents.

There are alternatives to the implementation of a new file system approach as we describe in the paper. 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.

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. We present here some examples of how this flexibility can be exploited by the user. 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.

2.2 System Design

UmbrellaFS functions as a stackable file system [30], 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 this paper, 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. Return values from the underlying file systems are likewise directed back to VFS, which then returns them to the application.

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 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.

The policy decision module is in many ways similar to a router in a network, directing traffic to the appropriate destination. Many of our 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 our system, the file is an analog of a packet and the destination file system is an analog of nexthop. While we have used the rules of first match and longest name match, other appropriate rules may be employed in other designs.

Figure 4 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 1 is based on access type and dictates that all read only files should be located on /fs2, the flash drive. 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.

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

Figure 4. Sample UmbrellaFS Rules

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 shrunk) 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 we evaluate later.

Such lazy enforcement of policy on file closing cannot be useful with certain rules based on time of access. For example, if we want a file to be stored on a slow device if it hasn't been accessed in five days, we 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.

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. We study the impact of this overhead in our 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 3.3 provided backup systems while rules 2 and 4 only allowed one choice in locating the files.

3. IMPLEMENTATION

UmbrellaFS is implemented as a loadable module for the 2.6 Linux kernel. We leveraged Unionfs 1.0.14 [29] in the development of UmbrellaFS. Unionfs's ability to combine underlying file systems and representing them in a single directory is clearly 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.

The modifications from Unionfs 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, whereas Unionfs is more focused on 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.

The most notable change from Unionfs in UmbrellaFS is the replacement of many of the linear search elements in Unionfs. Because Unionfs can have multiple files with the same name in separate branches, Unionfs searches through all the branches linearly to locate a file. UmbrellaFS on the other hand uses the policy decision module to directly access the correct branch.

Another significant change is the removal of whiteouts. Whiteouts are specially named files which were treated differently by Unionfs. If a whiteout for a given filename was present, then Unionfs did not search through the remaining branches for the file. This was used to hide underlying files when the file in the highest branch was deleted, or to allow for the apparent deletion of read-only files. Since there is only a single file in all of the underlying file systems for any given file name, both whiteouts and their maintenance were removed from UmbrellaFS.

4. EVALUATION RESULTS

All the benchmark tests were run on a Dell Optiplex GX620 with an Intel 3.2 GHz Pentium D processor running a Red Hat Fedora Core 3 Linux 2.6.9 kernel with 2 GB of RAM. 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. Each test was run at least 3 times, and the averages are provided. Ninety-five % confidence intervals were calculated using the Student-*t* distribution. 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.

4.1 Bonnie++

Bonnie++ [2] 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 Figures 5.

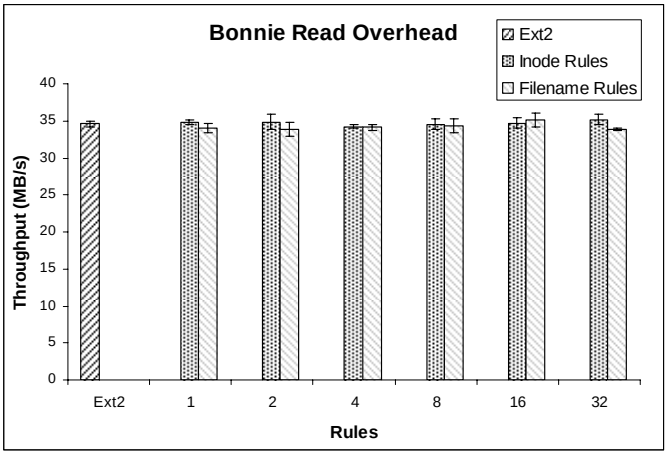

Figure 5a. Throughput in Bonnie++ benchmark for sequential reads

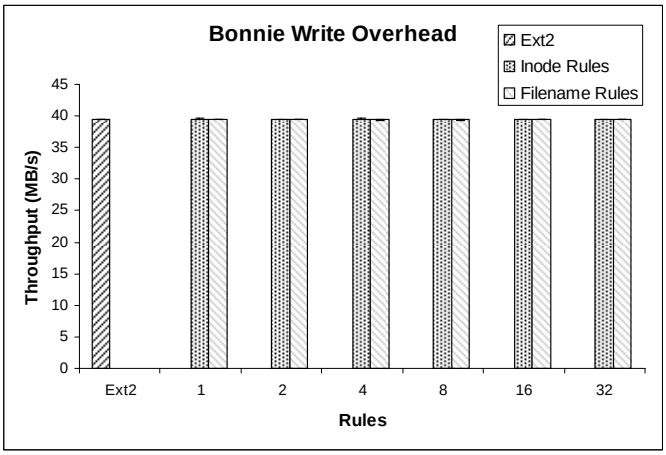

Figure 5b. Throughput in the Bonnie++ benchmark for sequential writes.

As can be seen in Figures 5a and 5b, 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. We can infer from this that there is not a large overhead associated with large reads and writes.

4.2 OpenSSH

Compiling OpenSSH [14] is a CPU-intensive benchmark [29]. We used OpenSSH 4.7p1. 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 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.

Figures 6 and 7 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, although given the relatively small amount of wait time, this did not have a large effect on the overall execution.

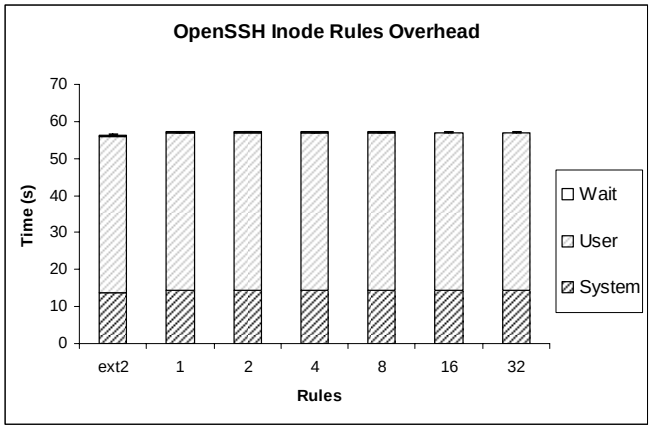

Figure 6. OpenSSH overhead with inode based rules.

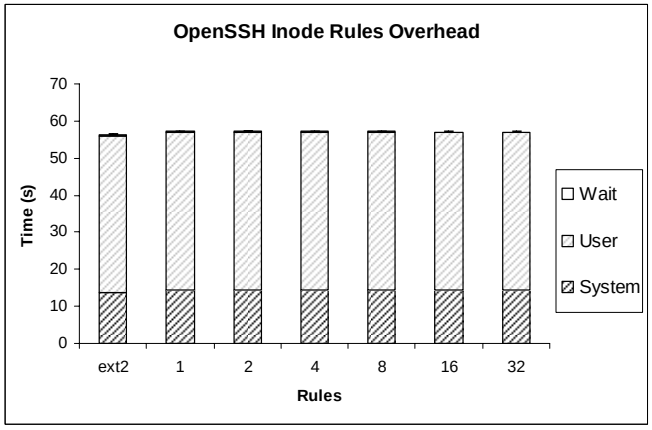

Figure 7. OpenSSH overhead with filename based rules.

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.

4.3 Postmark

Postmark [9] is an I/O intensive benchmark designed to simulate the operation of an e-mail server. In our Postmark tests, we created 12,500 files between 8 kB and 64 kB in 200 subdirectories and then performed 25,000 transactions.

Figure 8 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%.

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.

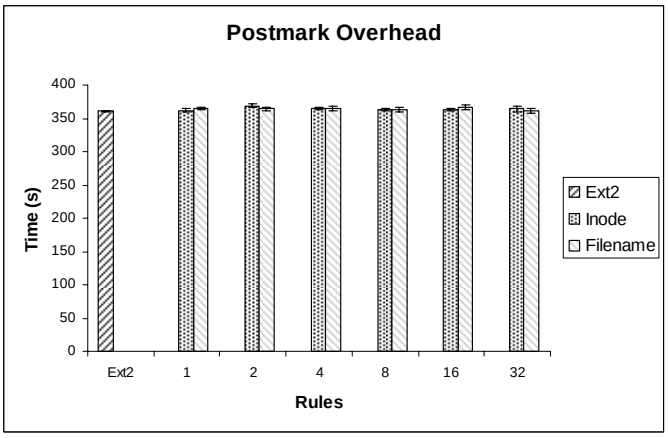

Figure 8. Postmark Overhead.

4.4 Overflow

Because the Postmark benchmark had the highest overhead with UmbrellaFS, we also used Postmark 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 10 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 9, 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.

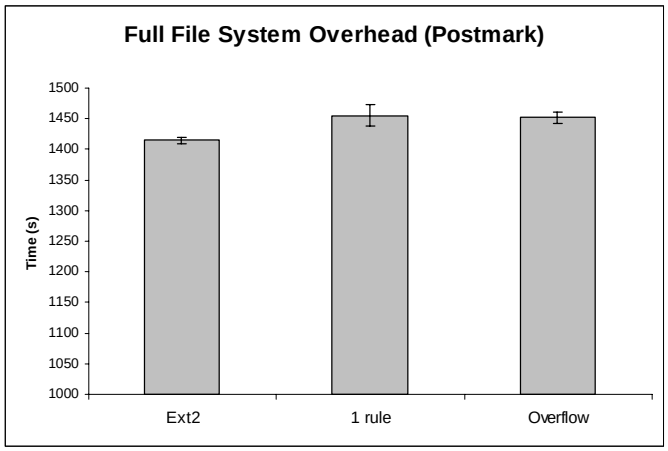

Figure 9. Postmark overhead with full file system

4.5 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, we used a micro benchmark which took a set of 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 10. 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.

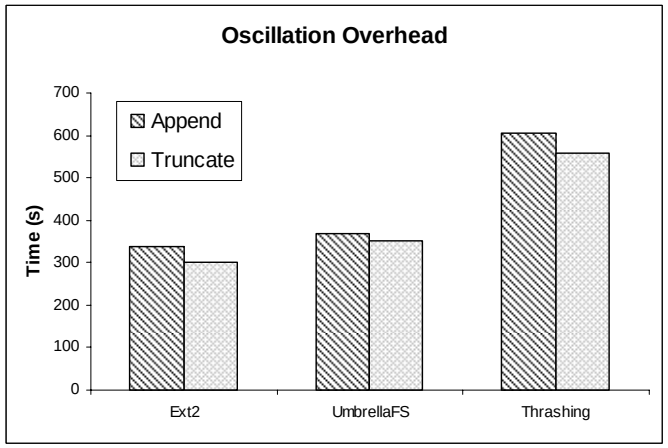

Figure 10. File size oscillation between 7.5 and 8.5 kB

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 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 179% on ext2 and 163% over UmbrellaFS without the file movement. Truncates showed an overhead of 187% and 159% 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 we are looking to include hysteresis to reduce this type of thrashing.

5. EXAMPLE UMBRELLAFS SCENARIOS

In this section, we demonstrate 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 (as explained earlier) and shows the possibility of a Coda [13] like disconnected operation.

5.1 Diverse Devices

In the first example scenario, we 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 11a and 11b.

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.

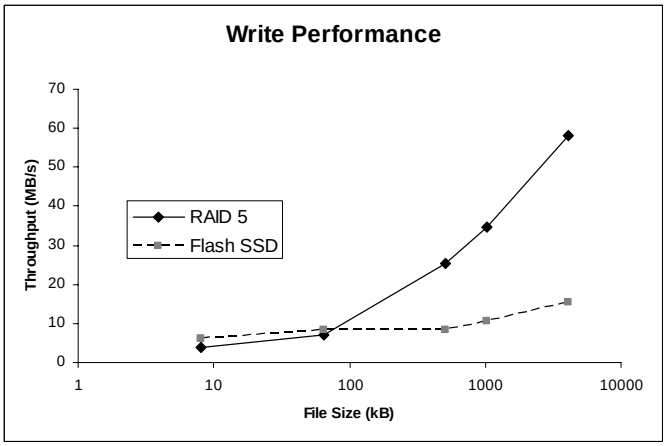

Figure 11a. RAID 5 and Flash SSD write performance

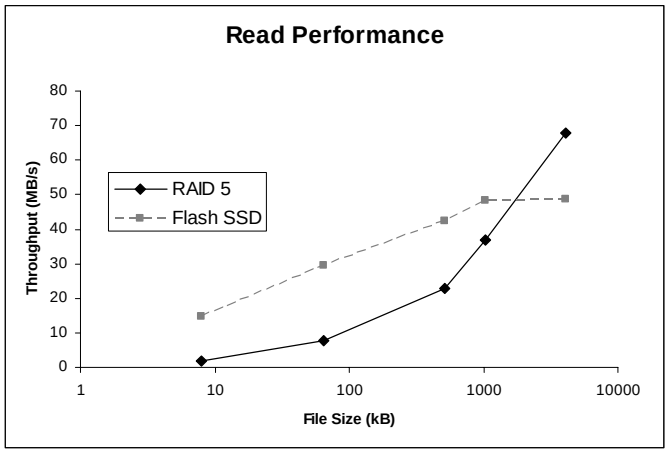

Figure 11b. RAID 5 and Flash SSD read performance

In order to mitigate the impact of limited number of write cycles of flash drives and the write performance gap at large file sizes, we 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. 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 12.

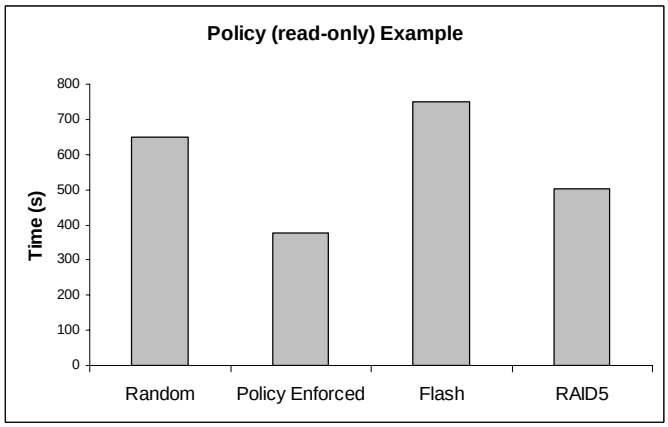

Figure 12. Policy Rule Example

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, we would not see the overhead of section 4.5, but rather the performance would have reverted to the RAID 5 curves in Figures 12a and 12b. We recognize that 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 [20].

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.

5.2 Encryption

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

We encountered difficulty 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, we utilized EncFS [12], an open source encrypted file system that makes use of the FUSE [22] library to run in user space. 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 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.

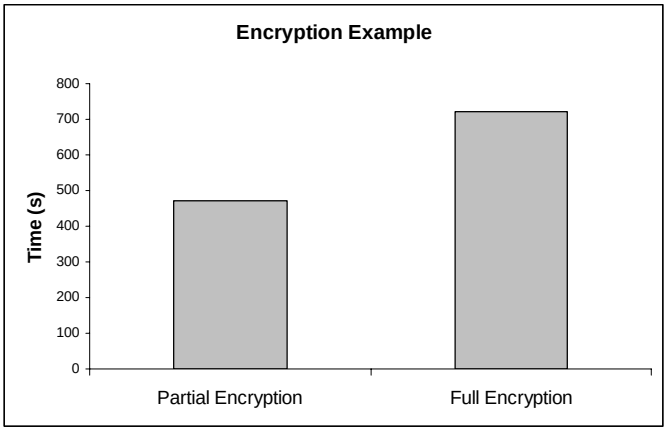

Figure 13. Encryption Example

Figure 13 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.

5.3 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), we utilize an AND based evaluation, we can 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, we rely on a feature of Unionfs, that is, the ability to add and remove underlying branches on the fly. By calling an appropriate ioctl, we can remove the branch that returned the error from UmbrellaFS, 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, we analyze the rules in the system, 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, we again utilize the Unionfs ability to add branches on the fly. Again we must evaluate the rules, and if an inactive rule's primary branch has returned, we reactivate that rule so that it again directs files to the newly returned drive. In this situation, however, we are faced with the files that have been moved to the local machine and need to be migrated back to the network storage. In order to accomplish this, we utilize a background program. Since our evaluation of a file and its correctness of location is done on open and close, we cannot move files 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, we can achieve disconnected operation similar to that provided by Coda, 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.

6. RELATED WORK

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 some of the same benefits without necessitating a total switch of file systems. UmbrellaFS can direct files to appropriate file systems mapped to the underlying device characteristics.

Duke's Active Names [25] 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.

HP's AutoRAID system is a device level approach to managing storage across RAID 1 and RAID 5 devices. In AutoRAID "hot" and "cold" data are stored in the RAID 1 and RAID 5 arrays, respectively. RAIF [8] allows striping of files across file systems.

Intel's Robson technology [24] 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.

Packet interposing [1] has been shown to provide benefits in scaling performance in a clustered file server.

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. Panasas's storage allocation policy scales the striping granularity based on the file size [15].

Brick based storage systems such as Self-* [4] and FAB [3] 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 [5] randomly distributes the blocks of a file across a number of nodes.

Sun's ZFS file system [21] 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 [28] is designed to span both persistent memory and magnetic disks. Conquest stores metadata and small files in memory, leaving large files on magnetic disk. UmbrellaFS can provide similar functionality in terms of the placement of files based on size. While UmbrellaFS cannot easily place metadata on one drive and data on another due to metadata consistency issues and the presence of the file systems on the underlying devices, UmbrellaFS has numerous additional rule possibilities in addition to Conquest's file-size threshold approach.

The Veritas File System [23] 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, either through direct action or through scheduling of the operation to examine the file system for files that are not correctly located based on policy.

7. CONCLUSIONS

In this paper, we presented the rationale and design of Umbrella File System. UmbrellaFS is designed to provide flexibility in mapping files to devices beyond the strict namespace bound mapping currently available. Our prototype implementation on Linux 2.6 showed that UmbrellaFS adds little overhead to most of the file system operations. We also demonstrated the utility of UmbrellaFS in both an example system organization consisting of a Flash drive and a RAID array, as well as a system utilizing selective encryption of files.

We are also looking into expanding the rule system in a number of ways. Notably we are looking at providing hysteresis to the system in order to minimize the negative effects of oscillation. We are also looking at 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.

Another major area that could be explored is that of replication in UmbrellaFS. At the moment, a file is only present in a single underlying file system. However, in some cases, particularly the disconnected operation presented in Section 5, it might be desirable to leave behind a copy of the file when the file is moved to a new file system, and then update that copy later with modified data. How to effectively maintain and operate on replicas in the system remains as future work.

ACKNOWLEDGMENT

This work is supported by a grant from NSF HECURA FSIO program and by funding from the Department of Defense.

REFERENCES

[1] Darrell C. Anderson, Jeffrey S. Chase and Amin M. Vahdat. Interposed Request Routing for Scalable Network Storage. In Fourth Symposium on Operating Systems Design and Implementation, 2000.

[2] Russell Coker. Bonnie++. *http://www.coker.com.au/bonnie++/*, 2001.

[3] Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence and Alistair Veitch. FAB: enterprise storage systems on a shoestring. In *Hot Topics in Operating Systems*, pages 133-138, Lihue, HI, May, 2003.

[4] Gregory R. Ganger, John D. Strunk, Andrew J. Klosterman. Self-* Storage: Brick-based storage with automated administration. *Carnegie Mellon University Technical Report CMU-CS-03-178*, August, 2003.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 29-43, October 2003.

[6] Michael Austin Halcrow. eCryptfs: An Enterprise-class Cryptographic Filesystem for Linux. In *Proceedings of the Linux Symposium*, 1:201-218, Ottawa, Ontario, Canada, 2005.

[7] Christoph Hohmann. CryptoFS. reboot.animeirc.de/cryptofs/, July 2007.

[8] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. RAIF: Redundant Array of Independent Filesystems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, September 2007.

[9] Jeffrey Katcher. Postmark: A New File System Benchmark. *http://www.netapp.com/tech library/3022.html*.

[10] S.R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Association: Summer Conference Proceedings*, pages 238-247, Atlanta, GA, 1986.

[11] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, Prashant Shenoy. Capsule*:* An Energy-Optimized Object Storage System for Memory-Constrained Sensor Devices. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor System*, pages 195-208, Boulder, CO, November, 2006.

[12] Valient Gough. EncFS Encrypted Filesystem. *http://arg0.net/wiki/encfs*, February 2006.

[13] J.Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In ACM SOSP, 1992.

[14] OpenBSD. OpenSSH. *www.openssh.org*, 2007.

[15] Panasas, Inc. Object Storage Architecture White Paper. www.panasas.com.

[16] David A. Patterson, Garth Gibson, and Randy H Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109-116, 1988.

[17] R.H. Patterson, G.A. Gibson, M. Satyanarayanan. Using Transparent Informed Prefetching (TIP) to Reduce File Read Latency. *Goddard Conference on Mass Storage Systems and Technologies*, pages 329-342, Greenbelt, MD, September, 1992.

[18] Samsung Semiconductor Europe. Samsung Flash Solid-State Drive. *http://www.samsung.com/eu/Products/Semiconductor/downloads/Samsung_SSD_for_mobile.pdf*. 2007.

[19] Seagate Technology LLC. Seagate Unveils New Giants -- 250GB Notebook Hard Drive and the First Encrypting 1TB Desktop PC Drive. *http://www.seagate.com/ww/v/index.jsp?locale=en-US&name=seagate-unveils-new-giants&vgnextoid=6bb0e0e1f0494110VgnVCM100000f5ee0a0aRCRD*. September 5, 2007.

[20] Prashant Shenoy, Pawan Goyal, and Harrick Vin. Architectural Considerations for Next Generation File Systems. *University of Massachusetts Technical Report UM-CS-1998-048*. October, 1998.

[21] Sun Microsystems. ZFS: the last word in file systems. *http://www.sun.com/2004-0914/feature/index.html*, September, 2004.

[22] Miklos Szeredi. Filesystem in Userspace. *http://fuse.sourceforge.net/*, retrieved November 2007.

[23] Symantec. 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,* retrieved November 2007.

[24] Michael Trainor. Overcoming Disk Drive Access Bottlenecks with Intel® Robson Technology. In *Technology@Intel Magazine*, 4(9), 9-11, December 2006.

[25] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal, Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, October 1999.

[26] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *ACM Transactions on Computer Systems*, 14(1):108-136, February 1996.

[27] Project T10. Information Technology—SCSI Object-based storage commands (OSD), Rev. 10, July 2004.

[28] A. Wang, G. Kuenning, P. Reiher and G. Popek. The Conquest File system: Better performance through a disk/persistent-RAM hybrid design. ACM Trans. On Storage, 2006.

[29] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, Erez Zadok, and Mohammad Nayyer Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Stonybrook University, October 2004.

[30] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. On Incremental File System Development. In *ACM Transactions on Storage*, 2(2):1-33, May 2006.