MVSS: Multi-View Storage System

Xiaonan Ma and A. L. Narasimha Reddy
Department of Electrical Engineering
Texas A & M University
College Station, TX 77843-3128
{xiaonan,reddy}@ee.tamu.edu

Abstract

MVSS is a storage system that offers a single framework for supporting a wide range of proposed new services. MVSS proposes to provide a flexible interface for associating services to a file through multiple views of the file. Multiple views of a file in MVSS are similar to views of a database in a multi-view database system. Multiple views of a file in MVSS are generated dynamically and are not stored on the physical storage device. MVSS represents each view of the underlying file through a separate entry in the file system namespace. Relative to approaches that provide services through namespace such as stackable file systems, MVSS separates the deployment of services from file system implementations and thus allows services to be performed at various levels of the storage system. This paper shows the design of MVSS and how various services such as encryption, network-attachment and application-level intelligence can be supported in MVSS at the device level. To illustrate our approach, we implemented an initial MVSS prototype on two PCs running Linux. The implementation requires no modification of the base operating system. We show how new services are developed and composed in MVSS. We present results from the prototype implementation to demonstrate the effectiveness of our approach.

1 Introduction

Many researchers have suggested various enhancements to disks to improve the performance, function and characteristics. These ideas can be characterized in the following way: (a) Device level enhancements such as compression have been used in the past, and enhancements such as encryption are being proposed. (b) Network-attached disks where data requested from the device is directly sent to the client rather than through the server [1, 2, 3, 4, 5]. Network-attached disks are shown to provide throughput scalability in file systems [3]. It is also expected that such third-party data transfers will improve server performance for new multimedia applications. (c) Active disks where part of the application is migrated and executed at the device resulting in "filtered" data [6, 7, 8]. It is expected that such "filtering" will improve performance of some applications such as decision-support applications by reducing the bandwidth demands on the I/O interconnect.

To enable such new functionalities, new host-disk interfaces have been proposed to accommodate higher-level services. This results in significant changes to the operating system and thus few of them have gained wide deployment in a timely fashion. Different approaches tend to have their own interface extensions, which greatly complicates and discourages third-party deployment and adoption of these extensions. A storage system built on the common block-level interface makes it easy to support a wide range of platforms.

Rigid device level enhancements may not be universally beneficial or acceptable. For example, It has been shown that enhancements such as network-attached disks do not uniformly benefit all applications. In some workloads, network-attached disks can reduce data cache hit rates sufficiently to more than compensate for the benefits of improved parallelism in network transfers [9]. A desirable storage system should allow users to efficiently adjust the system according to their particular requirements.

These studies have motivated the need for devices with more intelligence and function. It is likely that different enhancements may be proposed in the future. A number of other studies have motivated the benefits of migrating services to different levels in the system [10, 11, 12, 13].

Stackable file system model [14, 15, 16] has been suggested to provide services at file system (FS) level. However, in these systems, services can only be supported at the file system level. Some approaches associate services with the process of name resolution [11, 17]. Other approaches have built custom operating systems to support service migration [18].

Can we support service migration to device-level, file system level and above without significant modifications to existing systems? Can we design a flexible device interface and an accompanying storage system that accommodates many different enhancements? Can such devices be managed and accessed by current file systems without major modifications?

This paper tries to address these problems by proposing to build a Multi-View Storage System, MVSS. Similar to a Multi-View Database System [19], MVSS allows multiple views (also called virtual files in this paper) of the same file on the physical disk to be generated dynamically. Different views of a file can be customized to provide different types of service. MVSS separates the deployment of services from file system implementations and thus allows services to be performed at various levels of the storage system. Multiple views of the file are provided to the user through file system namespace. Through these views, MVSS provides a flexible and extensible way for supporting a wide range of services including device-level enhancements. We will concentrate on providing support for device-level services in this paper.

MVSS has the following novel combinations of characteristics:

- It uses common block-level interface widely used in today's systems. This allows it to support a wide range of heterogeneous platforms.
- It can be built on existing systems with little changes to the operating system. File system operations on normal files are not affected.
- It allows easy composition of new services from existing ones. Services could be loaded into or unloaded from the system dynamically.

- It provides a scheme to allow easy and secure migration of application-specified processing to devices to realize active disks.
- It allows applications to take advantage of new services transparently without significantly modifying the applications.

The rest of the paper is organized as follows. Section 2 presents the design rationale for MVSS. In Section 3, we describe some details about our prototype implementation and present the results from experiments. In Section 4, we compare various aspects of MVSS with related work. Section 5 concludes the paper and points to future work.

2 Design of MVSS

We describe the design of MVSS by examining the following key ideas of the system. First, MVSS introduces the concept of views of a file. A view of a underlying file, also called a virtual file, represents a combination of the file and certain services. Views of the same file could be different in content or behavior when associated with different services. Second, MVSS provides a flexible user interface to allow users to dynamically associate services with each view of a file. The interface also allows transparent service deployment. Third, service binding information (the services and parameters associated with the virtual file), is made available at the device level through a file-to-block level translation scheme that requires no change to the native file system. Fourth, a smart storage device model is employed, which can support a wide range of services using the common block-level interface. This is done by using block addresses that are beyond the physical capacity of the device.

In the following sections, we describe how those ideas are put together in developing MVSS.

2.1 Virtual Files

In MVSS, a virtual file provides a view of an underlying file (called base file) in the system. A view in MVSS is a general concept. A view represents a file associated with a certain service. The default view is the normal file system view of the underlying data. The provided view could specify a content-based service or behavior-based service. Examples of content-based views include "filtered", "compressed", or "encrypted" data. Examples of behavior-based views include third-party transfer and remote replication.

Figure 1 shows how virtual files and virtual disks relate to the underlying files and block devices. A virtual disk in MVSS is a generalized abstraction of a storage device. A virtual disk behaves like a normal block device to the rest of the OS but has no corresponding physical disk. Instead, it is "hooked" to an existing block device. Hooking a virtual disk causes all the IO requests sent to the virtual disk to be forwarded to the underlying device.

Mounting a virtual disk also creates virtual namespace for files on the device it is hooked to. For example, in figure 1, Three virtual disks (`/dev/vdisk1`, `/dev/vdisk2`, `/dev/vdisk3`) are hooked to directory `/foo` on disk `/dev/disk1`. Mounting these virtual disks then exports virtual separate views of all files descending from `/foo` under `/vd1`, `/vd2` and `/vd3` respectively. The three virtual

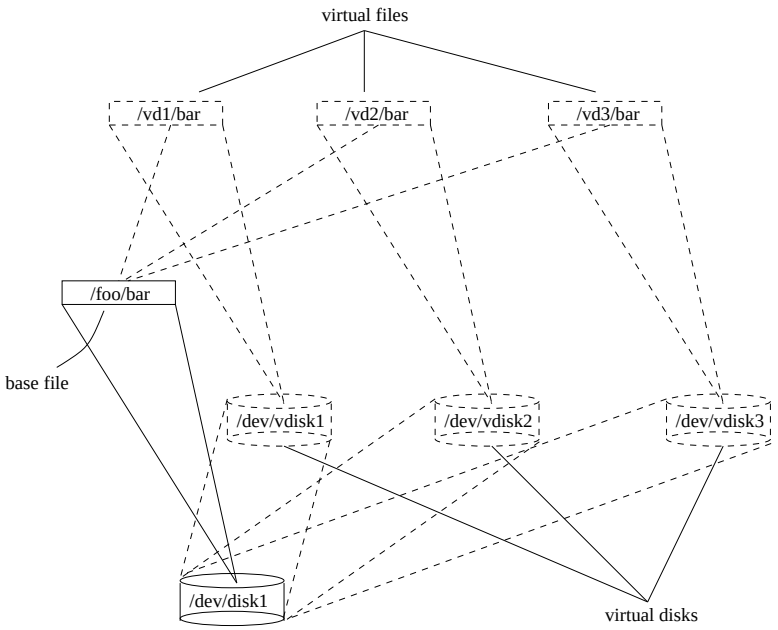

Figure 1: Concept of virtual files in MVSS

files `/vd1/bar`, `/vd2/bar` and `/vd3/bar` are different views of the same file `/foo/bar` on the physical disk. Virtual files look like ordinary files to the file system, but do not have any physical data blocks associated with them.

Caching is universally employed in today's file systems. Supporting multiple views of a file leads to the problem: different views of the file may contain different data, how should these views be cached? MVSS solves this problem by representing each virtual file as a separate file on a separate virtual disk. Each virtual file has its own pathname, in-core inode and uses separate buffers in the system. The caching problem is further complicated when writes are allowed on virtual files. Similar cache coherency problem exists for stackable file systems and is discussed in detail in [20].

Virtual disks facilitate namespace distinctions of different views of a file, provide a solution to the above discussed caching problem, and also allow service binding at the device level. We will discuss how virtual disks facilitate service binding in section 2.3.

Virtual files in MVSS are managed by MVFS, a stackable file system layer based on the vnode structure. Stackable file system structure is discussed in [14]. MVFS sits on top of native file systems and forwards file system operations such as name resolution to the native file systems.

2.2 File System Interface

A user-level interface is required to associate the virtual file with the user specified service. The interface should be flexible because different services may have different requirements.

MVSS provides a flexible interface — *attach* for binding services with virtual namespace entries. Attach has the following general interface: *attach(virtual file, service name, parameters)*. Service name specifies the service being provided, encryption, compression etc.. Examples of parameters include keys for encryption, QoS (Quality of Service) parameters for MPEG, query criteria for database

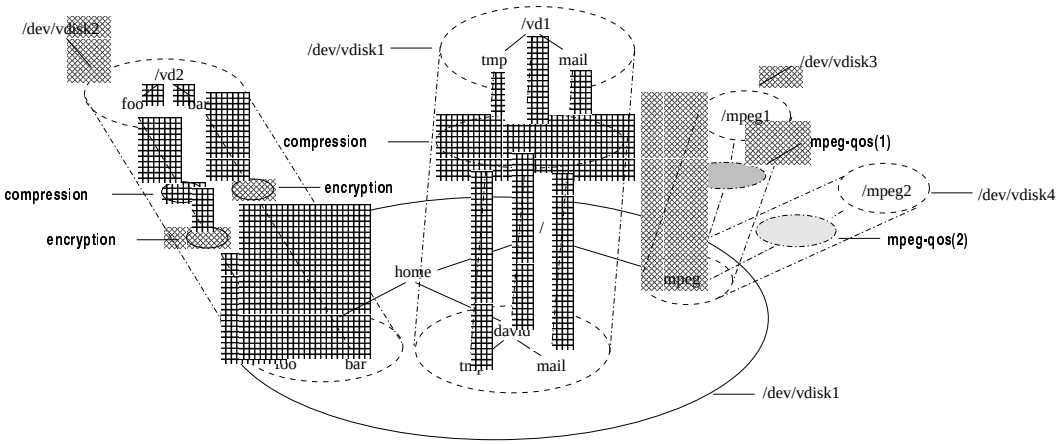

Figure 2: Example of service binding in MVSS

SELECT operation. Continuing with the earlier example, as a result of *attach(/vd1/bar, decryption, key)*, the virtual file `/vd1/bar` provides a decrypted (using the specified key) view, of the underlying encrypted file `/foo/bar`. The *attach* interface allows the service binding to be separated from other file operations such as open and read/write. This separation has the advantage that once the attach is done, following accesses automatically see the attached view. This allows transparent enhancements without modification of existing application codes.

The *attach* interface provided by MVFS supports binding at the file level for flexibility and also for more efficient use of the virtual name space. A single user may want to apply different enhancements to different directories. Attaching a directory affects all the files and subdirectories under it. If no service is attached to a virtual file directly, MVFS decides which service is to be bound to the virtual file by searching for the the right most attached point in its pathname. If none of the directories is attached, a default service will be associated with the virtual file.

Figure 2 shows an example of the service binding in MVSS. Virtual disk `/dev/vdisk1` is hooked to `/home/david` on `/dev/disk1` and mounted on `/vd1`. Attaching `/vd1` to a compression service allows all files under `/vd1` to be transparently compressed on disk. Virtual disk `/dev/vdisk2` is hooked to `/home/tim` and mounted on `/vd2`. File `/vd2/bar` is attached to an encryption service. `/vd2/foo` is attached to a compression-encryption service, which can be composed by stacking a compression service on an encryption one. Such a service allows files to be first compressed and then encrypted as they are written and the procedure is reversed as the files are read. Virtual disk `/dev/vdisk3` and `/dev/vdisk4` are both hooked to `/mpeg` on `/dev/disk1` and are mounted on `/mpeg1` and `/mpeg2` respectively. Both `/mpeg1` and `/mpeg2` are attached to an MPEG-transform service but with different quality parameters. Users accessing files under `/mpeg1` will see the same MPEG files under `/mpeg` but at quality level 1, and at quality level 2 under `/mpeg2`.

MVFS saves the service binding information (service name and parameters) for a virtual file in a data structure that is associated with each virtual inode, called *a area*[1]. The *a area* also contains

[1]The *a* in *a area* stands for "auxiliary".

information such as which virtual blocks on the virtual disk are allocated to the virtual file. Virtual block allocation will be covered later in this paper.

Sometimes the service parameters for a virtual file may need to be changed frequently. For example, services such as database filtering accesses require a new view to be generated when user changes the query specification. These views may be accessed at the same time. Allocating a namespace entry for each view in this case is not an efficient solution as a particular view like this may only be accessed once. In MVSS, *attach* allows user to change the service binding at any time. Because attaching a different service usually will produce a different view of the file, problem arises when there are still opened instances of the old view. To be consistent with the traditional UNIX file system semantics, MVSS allows the processes with the open handles to still see the old views. Processes accessed after attachment will see the new view. The reason for this is that attaching a new service to a virtual file is similar to replacing the old virtual file with a new one.

In our current implementation, a user is allowed to attach to a virtual file only if he is the owner of the underlying base file. This is done for security reasons, as allowing any user to attach to public shared files may result other users accessing unexpected data.

2.3 Service Binding at Device Level

The *attach* interface allows user to associate services with virtual files at the file level. The binding information needs to be passed down to device level, where the service is performed. The problem is that at the device level there is no concept of files. One possible solution is to develop new interfaces between the MVFS layer and the device driver instead of using the interfaces provided by the OS. However, doing this is similar to developing a new file system.

MVSS's solution uses the existing operating system's interfaces. Each virtual file in MVSS is allocated some virtual blocks, similar to how physical disk blocks are allocated to normal files in traditional file system. When a virtual file is attached, its *a area* is updated to reflect the new service. MVFS makes this binding information available at device level by associating all the virtual file's block numbers to the *a area* through a virtual block map. This is done efficiently in MVFS by managing virtual blocks in segments and using a dynamic virtual block allocation scheme. Details on this will be covered in section 3.2.

2.4 Device Model

The storage devices in MVSS are assumed to have enough resources to generate the specified views of the data. These resources usually consist of a processor with sufficient memory and a simplified embedded OS. These requirements are considered quite reasonable for future disks. Below, we describe some of the salient features of our device model.

2.4.1 Virtual Block Space

Devices in MVSS use the same common block-level interface as normal IDE/SCSI disks. MVSS solves the problems of binding service information with IO requests by making use of the *virtual block addresses* of devices. Virtual block addresses are block addresses beyond the physical capacity of the device.

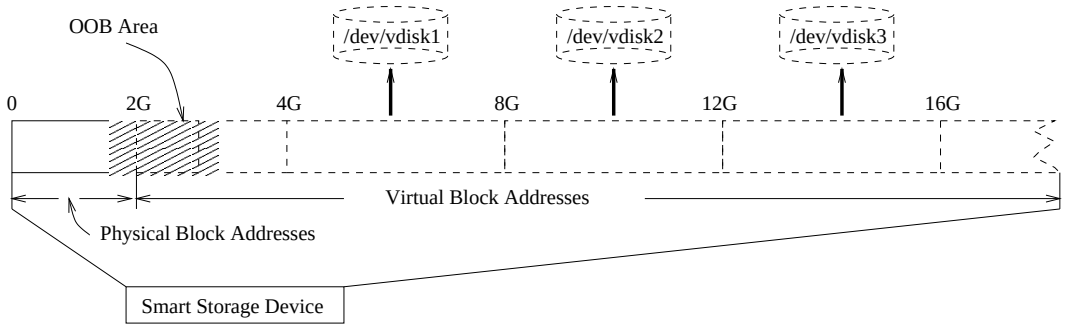

Figure 3: An example of virtual block address space management in MVSS

The idea is based on the following observation: Capacities of block devices are usually much less than the maximum value that the OS and device driver interface could support. For example, on a system that uses 32-bit integers to represent sector numbers on a block device could reference up to 2^{32} blocks, which means a capacity of 4TB with a block size of 1024 bytes. A similar approach has been recently adopted for building a flexible memory controller [21].

In MVSS, when a virtual disk is hooked to a device, it's allocated a portion of that device's virtual block address space. The size of the allocated block address range can be different from the target device's capicity. Block address ranges allocated to virtual disks hooked to the same device do not overlap with each other. Virtual addresses allocated to each virtual disk will be allocated to the virtual files on that virtual disk.

Each physical device in MVSS also reserves a certain range of virtual block addresses for "out-of-band" communication between the host and the device, called the OOB area. Messages are sent from host to the device by writing data blocks into the OOB area in a way similar to memory mapped IO. These messages contain information regarding filter loading/unloading and service binding. Through "out-of-band" communication, a smart storage device in MVSS keeps a synchronized copy of the related virtual block maps (one virtual block map for each virtual disk hooked to it) and all the *a area* of the virtual files being accessed. This allows the device to find out which service is associated with the requested data based only on the virtual block addresses.

Figure 3 shows an example of how a device's virtual block address space is managed in MVSS. In the example, a few virtual disks are hooked to a smart storage device, which has a physical capacity of 2GB. Each virtual disk is allocated 4GB of the device's virtual block space. For example, virtual block space from 4GB to 8GB is allocated to `/dev/vdisk1`. IO requests forwarded from a virtual disk to the smart storage device contain block numbers only within its allocated virtual block space. This enables the device to find out which virtual disk a particular IO request is comming from easily.

2.4.2 Programming Model

In MVSS, new services can be dynamically added or composed from existing ones. A new service is added by loading a piece of code — applet into the device. Applets could be in any format as long as the disk OS supports them, for example, they could be Java codes. There are mainly two types of applets: (1) system applets that can directly access the disk resources and thus should be carefully

written, as errors in them could result in corruption of the disk data. Only system administrators or authorized users should be able to load system applets onto the device. System applets may be specific to the host operation system or file system. (2) filter applets that contain codes to be invoked on the data that is being read from (written to) the device. Filter applets are similar to the stream modules in that: (a) filter applets get data from one end and produce output data on the other end. (b) filter applets can be pipelined, i.e., the output of one filter applet serves as the input of another filter applet.

Problem arises when output data from filter applets has a different size than that of the input data. For example, the number of records returned as a result of database select operation may not be exactly predictable at the time of initiating the query. Stream model is used in [6]. However, the stream model is designed for character devices and is inherently inconsistent with block devices interface. This is addressed in MVSS by restricting the input and output of filter applets to be in units of blocks. MVSS allows partial-valid data blocks to be returned to the host and MVFS prevents users from accessing the invalid data areas.

MVSS allows filter applets to be developed and loaded by normal users. Since the filter applets are executed at device level, without protection, a mistake in the code or a malicious user could easily corrupt the data or bypass the file system security schemes. MVSS provide a flexible and secure environment for filter applets by using the following mechanisms.

To prevent filter applets from crashing the disk OS, filter applets are executed at a process level on the disk, i.e., inside working processes. They are not allowed to access the disk resources directly and can only use the interfaces exported by the disk OS and system applets. Filter applets access the disk resources through interfaces provided by the disk operating system and system applets. Requests from the host are passed to processes through IPC mechanisms. There are two way of doing this: (a) A pool of working processes is available. When a request arrives, one of the working processes is selected to perform the filtering. (b) A dedicated process can be used to serve all the IO requests to virtual blocks that bound to the same virtual file. The former approach has less overhead and is suited for filter applets that are "stateless". The latter requires more management overhead but provides an easy programming model and allows "stateful" services to be deployed. "Stateful" services require state information to be maintained between individual requests.

Some filter applets may need to initiate IO requests on their own. Allowing a filter applet to access any blocks on the disk would break file system protection. In the Active Disk model [6], disklets are blocked from initiating IO requests and have to rely on host-resident codes to issue IO requests for them. MVSS solves this problem by allowing filter applets to initiate IO requests, but only to the data blocks belonging to the corresponding base file of the virtual file that the requested virtual blocks belong to. In this way, mistakes or malicious code in filter applets will not result in unauthorized access or harm the integrity of the file system and metadata on the disk. They can do no more damage than a normal user process that runs on the host accessing the same file. This requires supporting file block offset to physical block number conversion, i.e., the *bmap* interface on traditional UNIX file systems, on the disk side. This requirement won't increase the complexity of our device model too much. Our

prototype implementation shows that it takes only around 100 lines of C code to provide complete ext2 file system *bmap* support on the disk. The *bmap* function is file system dependent, but can be loaded into the device as system applets.

To prevent a computation intensive filter applet from monopolizing the processor on the disk, different priorities can be assigned to different working processes. Also the memory allocation for a filter applet can be restricted.

2.5 Applications

Below we list a number of example services that can be implemented in MVSS:

- Encryption

  Cryptographic techniques are playing an increasingly important role in modern computing system security, however, user-level tools are usually cumbersome. Adding cryptographic support at system level provides better transparency and performance. One such approach is presented in [22], which is built on the NFS interface. Secure storage at system level can be achieved in MVSS by using a filter applet that encrypts the data blocks on writes and decrypts it on reads. Keys are specified during the attach operation as parameters to the crypt applet. Different keys can be used for separate files and directories. Since the applet is executed at the device level, unencrypted information will be transmitted over the disk-host link, this problem can be solved by an MVSS implementation that uses its own encryption scheme to provide a secure disk-host link for sensitive information.

- Database Decision Support

  Decision support and data warehousing database workloads comprise and increasing fraction of the database market today, and the IO capacity and processing requirement grows rapidly. To meet this need, several several researcher have proposed the Active Disk/IDISK structure. Offloading computation closer to the data source offers benefits such as more computing power at little extra cost, potential reduction in data movement through the IO subsystem, better scalability, etc. These benefits can be achieved in MVSS by loading a database DSS applet onto the device that filters the dataset and only returns selected records. Similar data-intensive applications have been discussed in [23, 8, 7].

- MPEG QoS

  It is suggested in [11, 24] that the proper way to present web content to a particular client depends upon its individual characteristics. For example, it makes little sense to send a high quality MPEG stream to a hand-held device with a small black and white screen behind a wireless link. MPEG files are large enough that storing the same file at different levels of quality on the disk may not be an economical solution. In MVSS, filter applets can be developed to transform multimedia data to fit a particular client's requirements. For example, an MPEG applet can generate multiple views of an MPEG file at different levels of quality. The quality of

a view can be specified by supplying parameters such as frame rate, resolution, color at the time of the attach operation.

- QoS Scheduling

  Disk requests can be classified as periodic requests, interactive request and aperiodic requests and scheduled at the device level accordingly [25, 26]. This can be done in MVSS through a QoS scheduling applet on the disk. The QoS type of virtual file is specified during attaching. A QoS scheduling applet is an example system applet.

- Third-Party Data Transfer

  If the disk in MVSS has a network interface, then third-party transfer can be realized by letting the applet directly send the data back to the client. This allows bypassing the IO subsystem on the file server and will improve the performance in certain workloads [3, 9]. Information such as network addresses and port numbers can be supplied to the applet as parameters. The advantage of doing this in MVSS is that users can specify which files should have third-party transfer enabled. For example, in some workloads, it is better to return the data to the file server to exploit server side caching. While in other cases (e.g., a sequentially accessed video file), a direct transfer to the client may be beneficial.

- Other Potential Applications

  Device-level data replication, automatic data backup, remote disaster recovery backup are some of the other possible applications that can be implemented in MVSS.

3 Implementation and Results

In this section, we describe some of the details specific to our prototype implementation. Following goals have driven the implementation of MVSS:

- Keep the operating system layer on the disk as thin as possible. This allow efficient use of the disk resources and keeps the cost low. For example, requiring the disk to have full file system functionalities would essentially turn the disk into a mini file server and render those enhancements to be limited to that file system.

- Allow the disk to support a wide range of services. Supporting more services often requires more OS support on the disk. At the same time, we want to retain as much management work as possible on the host. This simplifies the disk model by taking advantage of the resources on the host side.

- Minimize changes to existing operating system's interfaces.

Table 1: Test-bed System Configuration

Host Machine Processor	166MHz
Disk Transfer Rate on Disk Machine	7MB/s
Smart-Disk Processor	166MHz
Host-Disk Interconnection B/W	10Mbits/s

3.1 System Configuration

Our MVSS prototype is built on two 166MHz PCs running Linux. One of them acts as the smart-disk, another as the host. The device is simulated by loading a slightly modified network block device (NBD) driver on the host and running a user space daemon on the smart-disk. We use a 1G byte partition on one of the smart-disk's internal SCSI disks as the raw disk. The daemon on the smart-disk bypasses the file system on the disk and performs IO operations directly on the device file. This is done to simulate a "smart disk" that may be available in the future. Table 1 shows the characteristics of our test-bed.

On the host side, both the MVFS file system layer and the virtual disk device driver are implemented as loadable kernel modules. We also developed a user space daemon for managing the *a area*s and virtual block allocation in user space. This was done for simplicity. The user space server is only invoked when certain operations are being applied to the virtual file, such as open, close and allocation of virtual blocks. The experimental results show that the overhead of moving part of the work into user space is acceptable.

Our current implementation of MVFS is read-only, we plan to implement write support in the future. Figure 4 illustrates the system structure of our prototype implementation. IO requests to virtual blocks are forwarded to the NBD driver by the virtual disk device driver. IO requests to both normal disk blocks and virtual disk blocks and "out-of-band" communication all go through the common host-disk interface provided by the NBD driver. They are then demultiplexed on the disk machine based on the block numbers and processed separately.

3.2 Virtual Block Allocation

One of the problems we encountered during the developing of MVSS is that how to efficiently translate the file level binding information into block level format. One straight forward approach wound be to just map the physical block allocation used in the native file system to virtual block space. Unfortunately, this approach doesn't lend itself to easy and efficient translation of binding information.

Binding all the blocks in a big file may require a number of disk IOs for meta data. The space requirement for storing the block map would also be prohibitive. For example, if the system uses 4 bytes to represent *a area* IDs, a block map for all the blocks on a 4G disk with block size of 1K bytes would have a size of at least 16M bytes.

When a virtual file is opened, we may not know ahead of time how many data blocks may be present in that virtual file. For example, the number of records returned as a result of database select

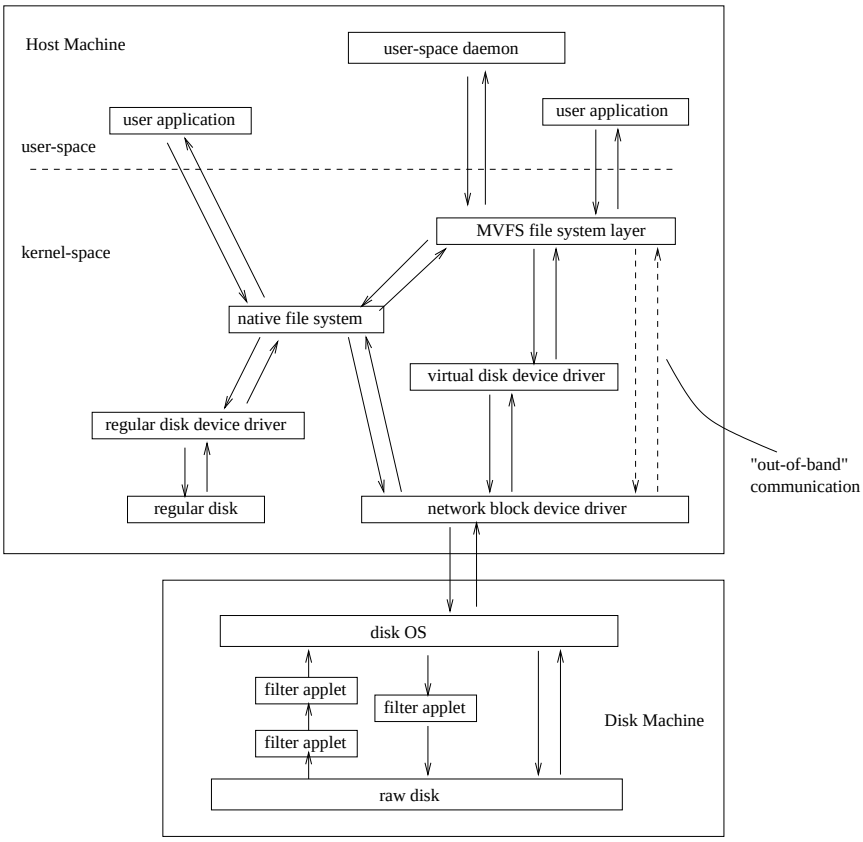

Figure 4: System structure of our MVSS prototype

operation may not be exactly predictable at the time of initiating the query. This is another reason for allocating virtual blocks dynamically.

To solve this problem, a dynamic virtual block allocation scheme is used in MVSS. We chose to manage the virtual block numbers in segments. Each virtual block segment represents a number of continuous virtual blocks. Virtual block segments are only allocated to virtual files when needed, and can be allocated to other virtual files once the virtual file is no longer in use. To make maximum use of the system buffer cache, MVSS always try to allocate the same virtual segments to a virtual file when it's accessed again. If there are cached blocks belonging to a segment, they need to be flushed from the buffer cache before the segment could be allocated to other virtual files. A good virtual segment cache scheme and a garbage collection process can help to reduce the overhead to a minimum.

To allow efficient block allocation for virtual file with different sizes, MVSS divided the whole map into a number of zones. Virtual block segments in different zones have different sizes. Both segment sizes and number of zones can be adjusted according to user's needs. Figure 5 shows how the virtual block space of a virtual disk is managed through a virtual block segment map. For example, each map entry in the first map zone represents 1024 continuous virtual blocks while each entry in the second zone represents 32768 continuous virtual blocks. The content of each map entry contains a caching status flag and reference to the *a area* that all the virtual blocks in side the segment are associated

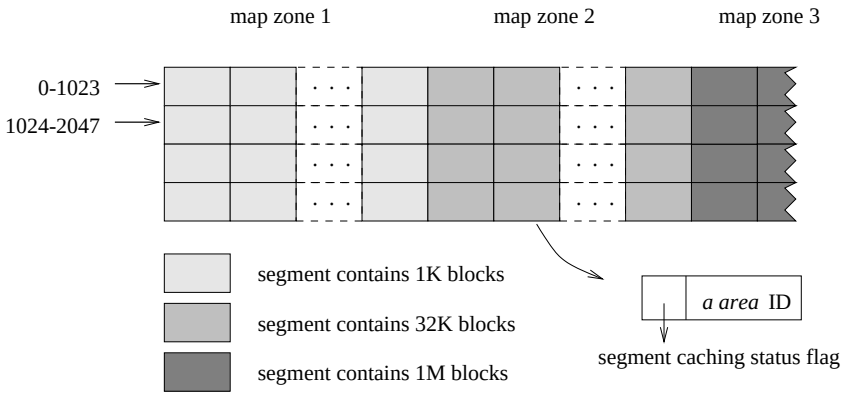

Figure 5: Example of virtual block segment map

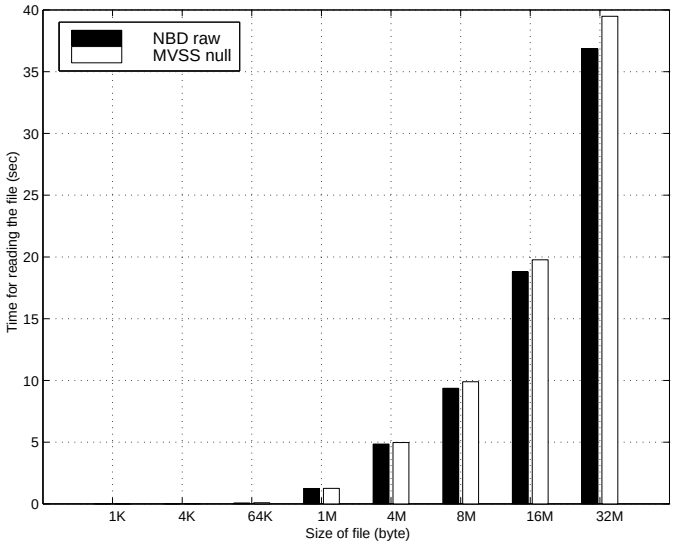

Figure 6: Overhead of MVSS with null applet

with.

The dynamic virtual block allocation scheme allows binding of all the virtual blocks of a virtual file to be done by just changing a few entries in the map without reading any meta data from the disk. It also reduces the total size of the map significantly, for example, in our implementation, the segment map for a 1G disk only takes 8k. Maps of this size could be stored in the memory of the disk easily.

3.3 Results and Analysis

Figure 6 shows the overhead of reading virtual files attached with the null applet in our prototype implementation. The null applet does nothing but copying the input data to the output buffer. The overhead mainly includes the time for translating the virtual block numbers into physical block numbers on the disk, and the time for extra buffer allocation, plus the overhead of FS system stacking and MVFS meta data management on the host. The results show that our implementation adds about 8% overhead for accessing virtual files compared to normal files. It is expected that the overhead could

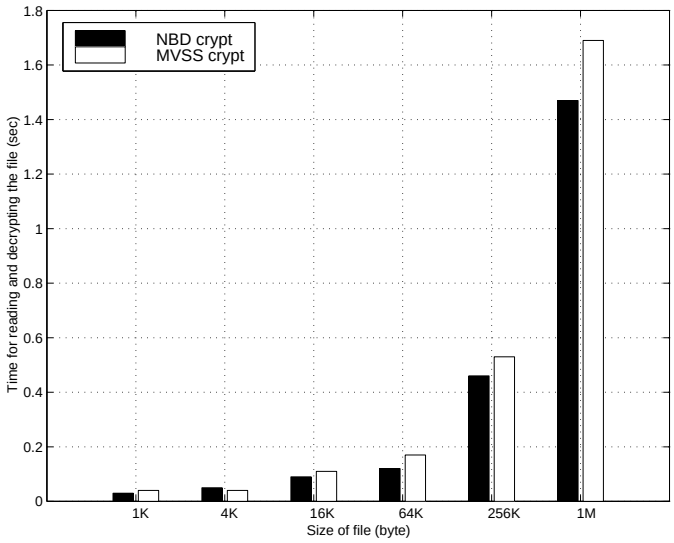

Figure 7: MVSS performance with crypt applet

be further reduced after further optimizations. we move virtual block space management into kernel space.

Three filter applets are developed to demonstrate the power of MVSS: encryption, database selection and MPEG. Encryption shows the flexibility of the user level interface. Database selection and MPEG demonstrate the performance gains by pushing processing to the disk.

Table 2: Complexity of Example Applets

Filter Applet	Lines of C codes
Encryption	185
Database Selection	90
MPEG QoS	220

All of these applets are written in C, table 2 gives the size of the applets in number of lines of C code. It shows that MVSS can greatly reduce the development and maintenance effort. MVSS makes it practical for normal end users without knowledge of the system internals to apply device level enhancement according to their needs. A host-resident developing environment (for example, a disk applet compiler) can further simplify the development procedure by providing a portable abstract interface for disk resources.

- Crypt

  A crypt filter applet is created by using the algorithm in the Unix crypt utility at block level. The practical value of this implementation is quite limited. The main purpose for it is to show that useful filter applets can be developed with the same level of effect required in developing user level applications. In simplified implementation, the hashed key is stored in the virtual

file's *a area*. Similar to CFS [22], access to attached virtual files is controlled by restricting the virtual directories through the standard UNIX file protection mechanism. To prevent the super user from reading the decrypted virtual file, a flag is introduced for virtual files in MVFS, which can be set at attach operation to indicate that access to the virtual file or directory should be restricted to the virtual file owner.

Figure 7 shows the performance of MVSS with the crypt applet (MVSS crypt) compared with a user application that reads the encrypted file and then decrypts it on the host (NBD crypt). The results show an overhead of around 10% for MVSS similar to what is observed with the null applet.

- DATABASE Selection

  To show the benefits of moving processing to the data source, we created datasets with 64-byte records. A small filter applet is developed to filter records based on a user-specified predicate, which is passed to the applet as a parameter. This simulates non-index SELECT operations that require scanning the entire dataset. The results obtained by specifying different selectivity on datasets with different sizes are shown in figure 8(a). A selectivity of N means that out of every N records in the original dataset, one record is selected and put into the output buffer. Reading from the virtual files shows a small amount of overhead when selectivity is 1, this is consistent with the results in figure 6. Figure 8(b) shows MVSS gains a performance increase or speedup of about 3.5 fold at selectivity 4 and 7 fold at 64. In our system the I/O interconnect throughput is the bottleneck compared to the disk IO bandwidth. However, as the selectivity increases, the demand on the interconnect bandwidth reduces, making the disk IO time the bottleneck. As a result, the speedup does not scale with higher selectivities (without a corresponding increase in disk bandwidth). Figure 8(c) shows that the time differences are proportional to the bandwidth reductions. Figure 8(d) shows that with high selectivity, the performance approaches that obtained by reading and filtering data directly on the disk machine. The results show the benefits of MVSS over a traditional disk when the disk to host IO interconnect is the bottleneck.

- MPEG

Table 3: Results for the MPEG applet

MPEG File Size (byte)		Time (second)			CPU Usage (%)	
Original Size	Filtered Size	Raw NBD Read	NBD with Filtering	MVSS with Filtering	NBD	MVSS
16M	6M	19.04	28.37	17.44	52	3
32M	13M	36.85	57.10	35.23	51	3

An MPEG filter applet is developed to demonstrate that a smart disk can be turned into an MPEG aware disk easily. Table 3 shows the results obtained by applying a MPEG filter applet on the video streams in MPEG files to throw away data for B and P frames. We have also implemented a finer level MPEG filter that discards slices of picture frames instead of the entire

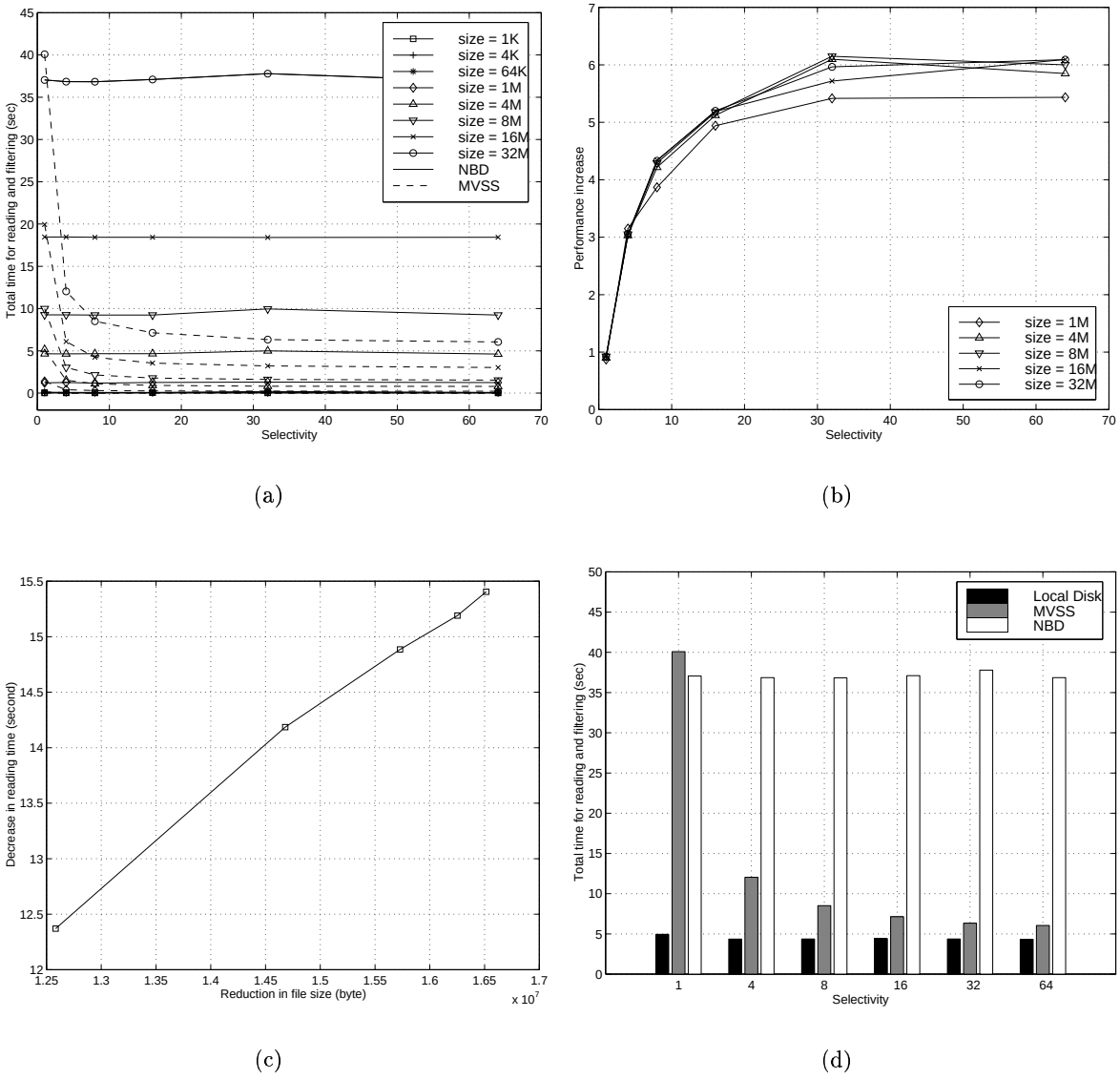

Figure 8: Results with simple database selection filter applet

frames. In the table 3, we show three cases of reading the MPEG data. Raw NBD read shows the time taken for reading the whole MPEG file over to the host without any processing. NBD with Filtering shows the time taken for reading the whole MPEG file over to the host and for filtering the video stream on the host. MVSS with Filtering shows the time taken for filtering the data on the disk through the MPEG filter applet. The results show a speedup of 40 percent. Also the CPU usage on the host drops from 50 percent to 3 percent as the processing is now pushed to the disk.

4 Related Work

Many approaches have been developed to provide flexible storage systems. One way to provide more flexibility in file systems is through Vnode Stacking [14, 15, 16]. These techniques allow the interposition and composition of vnodes so that file system modules could be layered on top of each other. StackFS [27] and FiST [28] are also based on this concept. Typical examples of stackable FS includes cryptfs [28]. The advantages of this scheme includes performance (as implemented in kernel) and user transparency. MVSS provides a more flexible way of binding services and also allows service migration to the device.

Another way to add flexibility on top the native storage is through the NFS protocol. Examples of this idea include CFS [22], Working Version File System [29]. This approach offers great flexibility and requires no change to the operation system and the native file system since it could be completely implemented in user space. However, this comes at the cost of requiring extraneous data copies. It is possible to modify current NFS protocol to talk to a server at different ports. The file server could implement a different filter at each network port. However, such an approach may not harness the possible parallelism across multiple storage devices and would not reduce the I/O bandwidth demands by filtering data close to the devices. Such an approach would also be specific to the NFS file system.

Virtual disks [30] and logical disks [31] have been proposed to improve storage organizations and file systems. Virtual disks in MVSS are a different generalized abstraction of a storage device.

Safety during kernel extensions has been addressed in [32, 33]. Exokernel allows customization of operating system functions through loadable OS library modules [34]. Code mobility [35], and protection [36, 32, 33] issues are not addressed here.

In response to the increasing storage and computational demand for applications such as decision support database, multimedia, active Disk/IDisk models [37, 6, 7, 8] have been proposed. These models propose to take advantage of the processing power on individual disks to run application level code. Analytical models and prototype simulators of active storage have been developed. MVSS draws much inspiration from this work. The results reported here are based on a real implementation. MVSS supports a block-level interface unlike the streams model proposed earlier.

In HURD [38], file systems are implemented at the user level. HURD has the concept of a translator, which is a program that can be attached to a pathname and perform specialized services when that pathname is accessed. Originally, a translator-like idea was used in Alex [39] to allow transparent ftp access via a file system interface. Other attempts to extend file system functionality include

"watchdogs" [40] and the Intensional File System [41]. Both provide some mechanism for trapping file system operations and allowing user specified arguments. The idea of extending file system services through name space is further generalized in Active Names [11].

A number of approaches for supporting computation migration are proposed in Active Names [11], Active networks [12], Sprite [42], Active Messages [43], Liquid software [10, 18], Derived Virtual Devices [44, 13]. Most of these approaches allow service migration only above the file system.

5 Conclusions

We have proposed to use multiple views of storage to build a flexible storage system. The multiview storage system, MVSS, provides multiple views of an underlying file, similar to the multiview database systems providing multiple views of the underlying database. We showed that the block-level interface afforded by MVSS allows flexible service deployment within existing file systems without any significant changes to the underlying OS. We also showed that it is possible to build MVSS without porting significant amounts of file system functionality on to the device. Results from a Linux PC-based prototype system demonstrated the effectiveness of MVSS.

Though our current implementation focused on supporting services at the device level, the idea of using views to combine files with services/behaviors can be applied at other levels of the storage system. In the future, we plan to generalize MVSS into supporting services at various levels of the storage hierarchy. We also plan to port more services onto MVSS to demonstrate the flexibility of the approach.

References

[1] Seagate Corporation. Fibre channel: The digital highway made practical. Technical report, Seagate Corporation, 1994. http://www.seagate.com/support/disc/papers/fibp.shtml.

[2] Robert W. Horst. TNet: A reliable system area network. *IEEE Micro*, 15(1):37–45, Feb. 1995.

[3] Garth A. Gibson and et al. File server scaling with network-attached secure disks. In *Proceedings of the Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Seattle, June 1997.

[4] R. Van Meter. A brief survey of current work on network attached peripherals(extended abstract. *Operating Systems review 30,1*, Jan. 1996.

[5] D. A. Menasce et al. An analytic model of heirarchical mass storage systems with network-attached storage devices. *Proc. of SIGMETRICS*, May 1996.

[6] A. Acharya, M. Uysal, and J. Saltz. Active disks. *Proc. of ASPLOS Conf.*, Oct. 1998.

[7] E. Riedel, G. Gibson, and C. Faloustos. Active storage for large-scale data mining and multimedia. *Proc. of 24th VLDB Conf.*, 1998.

[8] K. Keeton, D. A. Patterson, and J. M. Hellerstein. The Intelligent Disk(IDISK): A revolutionary approach to database computing. *Tech. report, Univ. of Cal., Berkeley*, 1998.

[9] Gang Ma and A. L. Narasimha Reddy. An evaluation of storage systems based on network-attached disks. *Proc. of ICPP Conf.*, Aug. 1998.

[10] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid software: A new paradigm for networked systems. 1996.

[11] Amin Vahdat, Thomas Anderson, and Michael Dahlin. Active names: Programmable location and transport of wide-area resources. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[12] Albert Banchs, Wolfgang Effelsberg, Christian Tschudin, and Volker Turau. Multicasting multimedia streams with active networks. Technical report 97-050, International Computer Science Institute, 1997.

[13] R. Van Meter, G. Finn, and S. Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. *Proc. of 8th ASPLOS*, Oct. 1998.

[14] D. S. H. Rosenthal. Requirement for a "stacking" vnode/vfs interface. In *Unix International document SD-01-02-N014*, 1992.

[15] G. C. Skinner and T. K. Wong. "stacking" vnodes: A progress report. In *USENIX Conference Proceedings*, pages 161–74, 1993.

[16] J. S. Heidemann and G. J. Popek. File system development with stackable layers. In *Transactions on Computing Systems*, 1994.

[17] M. Fouts, T. Connors, S. Hoyle, B. Sears, T. Sullivan, and J. Wilkes. Brevix design 1.01. *HP Labs Tech. Report, HPL-OSR-93-22*, Apr. 1993.

[18] J. Hartman, L. Peterson, A. Bavier, P. Bridges, B. Montz, R. Pilz, T. Proebsting, and O. Spatscheck. Joust: A platform for liquid software. *IEEE Computer*, 1999.

[19] J. O. Ullman. Principles of database systems. 1984. Computer Science Press.

[20] J. S. Heidemann. Stackable design of file systems. Ph.d. dissertation, University of California, Los Angeles, Los Angeles, September, 1995.

[21] J. B. Carter et al. Impulse: Building a smart memory controller. *Proc. of 5th Int. Symp. High Performance Computer Architecture*, Jan. 1999.

[22] Matt Blaze. A cryptographic file system for unix. In *First ACM Conference on Communications and Computing Security*, 1993.

[23] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Structure and performance of decision support algorithms on active disks. 1998.

[24] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. *Proc. of 7th ASPLOS*, pages 160–170, Oct. 1996.

[25] Reference omitted for anonymity.

[26] P.J.Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. *Proc. of ACM SIGMETRICS*, June 1998.

[27] CrosStor. The crosstor stackfs architecture. On-line White Paper, CrosStor Software, Inc. URL: http://www.crosstor.com/Technology/StackFSWP.html.

[28] Erez Zadok. Cryptfs: A stackable vnode level encryption file system. Technical report, Computer Science Department, Columbia University, 1998.

[29] Working Version. Wvfs - version-control filesystem. On-line White Paper, Working Version, Inc. URL: http://www.wv.com/hf-white-paper.html.

[30] C. R. Atanasio, M. Butrico, C. A. Polyzois, S.E. Smith, and J.L.Peterson. Design and implementation of a recoverable virtual shared disk. *IBM Technical report no. RC 19843*, Nov. 1994.

[31] W. de Jonge, M. F. Kasshoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. *Proc. of 14th ACM Symp. on Operating Sys. Principles*, pages 15–28, Dec. 1993.

[32] B. Bershad et al. Extensibility, safety and performance in the spin operating system. *Proc. of ACM SIGOPS*, Dec. 1995.

[33] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Surviving misbehaved kernel extensions. *Proc. of 2nd OSDI*, Oct. 1996.

[34] M. F. Kaashoek et al. Application performance and flexibility on exokernel systems. *Proc. of 16th SOSP*, Oct. 1997.

[35] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, pages 213–239, Sept. 1997.

[36] R. Wahbe, S. Lucco, and T. Anderson. Efficient software-based fault isolation. *Proc. of 14th ACM Symp. on Oper. Sys. Principles*, pages 203–216, Dec. 1993.

[37] J. Gray. What happens when processing, storage bandwidth are free and infinite? *Keynote speech at IOPADS '97*, 1997.

[38] *http://www.debian.org/ports/hurd/hurd-doc-translator.*

[39] Vincent Cate. Alex - a global filesystem. In *Proceedings of the File Systems Workshop*, 1992.

[40] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the unix file system. In *USENIX Conference Proceedings*, pages 267–75, 1988.

[41] P. R. Eggert and D. S. Parker. *File systems in user space.* Usenix Association, Berkeley, CA, 1993, 1993.

[42] F. Douglis. Experience with process migration in sprite. In *Proceedings of the First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, pages pages 59–72, October 1989.

[43] Thorsten von Eicken, Seth Copen Coldstein David E. Culler, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *In Proceedings of the 19th International Symposium on Computer Architecture*, pages pages 256–266, May 1992.

[44] R. Van Meter, S. Hotz, and G. Finn. Derived Virtual Devices: A secure distributed file system mechanism. *Proc. 5th NASA Conf. on Mass Storage Systems and Technologies*, Sept. 1996.