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

This paper presents MVSS, a storage system for active storage devices. MVSS offers a single framework for supporting various services at the device level. It provides a flexible interface for associating services to a file through multiple views of the file. Similar to views of a database in a multi-view database system, views in MVSS are generated dynamically and are not stored on physical storage devices. MVSS represents each view of an underlying file through a separate entry in the file system namespace. MVSS separates the deployment of services from file system implementations and thus allows services to be migrated to the storage devices. The paper presents the design of MVSS and how different services can be supported in MVSS at the device level. To illustrate our approach, we implemented a prototype system on PCs running Linux. We present results from the prototype implementation to demonstrate the effectiveness of our approach.

1 Introduction

Many researchers have suggested various enhancements to storage devices to improve the performance, function and characteristics by migrating services to disks. The ideas can be characterized in the following way: (1) Device level enhancements such as compression have been used in the past, and enhancements such as encryption are being proposed. (2) Active disks where part of the application is migrated and executed at the device resulting in "filtered" data [1, 15, 11]. 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 on host, better scalability, etc. (3) Network-attached disks where data requested from the device are directly sent to the client rather than through the server [8, 6, 3, 13]. Network-attached disks are shown to provide throughput scalability in file systems [6]. It is also expected that such third-party data transfers will improve server performance for new multimedia applications. However, rigid device level enhancements may not be universally beneficial or acceptable. For example, it has been shown that 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 [12].

In traditional storage systems, storage devices provide a block-level interface. File system accesses data on the devices through block addresses. Flexible service migration to the storage devices usually requires passing more information to devices than what traditional file systems allow today.

A number of different approaches have been proposed to solve this problem. In Joust [10], custom operating system has been built to support service migration. Other approaches such as active disks proposed different host-disk interfaces to accommodate higher-level services. These approaches require substantial modifications to existing file systems. Few of them have gained wide deployment in a timely fashion because of the following: (1) file systems are large and extensions to them are difficult to maintain, (2) modifications and extensions may not be acceptable for commercial systems, and (3) different approaches tend to have their own interface extensions.

This paper introduces the multi-view storage system (MVSS). MVSS offers a single framework for accommodating migration of different services to active storage devices based on existing file system and disk interfaces. Multiple views of the file are provided to the user through file system namespace. Different views of a file can be tailored to provide different types of service. Through these views, MVSS provides a flexible and extensible way for supporting various device-level enhancements.

MVSS has the following combination of characteristics:

- It uses the common block-level interface widely used in today's systems. This allows it to support a wide

range of heterogeneous platforms, and allows the simplest reuse of existing file system and operating system technology.

- It provides a scheme to separate the deployment of services from file system implementations and thus allows migration of application-specified processing to devices to realize active disks.
- 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 applications to take advantage of new services transparently.

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

The major design objective of MVSS is to enable migration of services to the storage device within the existing file systems. We followed the following principles while designing MVSS: (1) keep the software layer on the disk as thin as possible to allow efficient use of disk resources, and (2) minimize changes to existing operating systems.

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 (i.e., virtual files) and virtual disks. A virtual file represents a combination of the file and certain services. Virtual disks are place holders for virtual files. 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, a smart storage device model based on the common block-level interface is employed in MVSS. Fourth, MVSS makes service binding information for each virtual file available at the device level through virtual block addresses.

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

2.1 Virtual Files and Virtual disks

In MVSS, a virtual file provides a view of an underlying file (called base file) in the system. Virtual file in MVSS is a general concept. It represents a file associated with certain services. Examples of services include encryption, compression and application-specified processing (e.g., a base MPEG file may have multiple views corresponding to different levels of quality), etc.

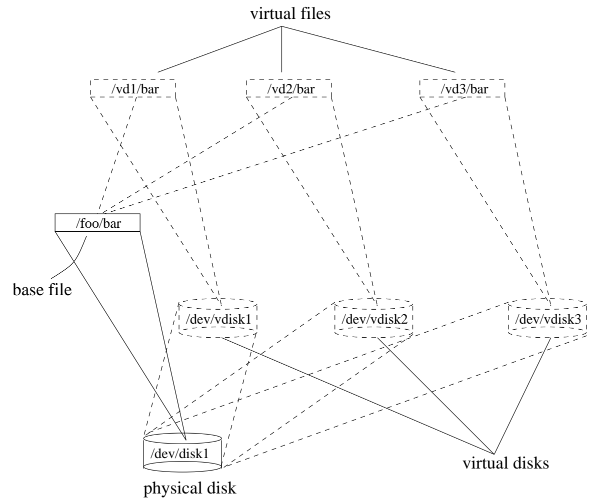


Figure 1. Concept of virtual files in MVSS

Most of the file systems today employ caching to improve performance. Supporting multiple views of a file leads to the following 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 disk. Each virtual file has its own pathname, in-core inode and uses separate buffers in the system. To support this, MVSS introduces the concept of virtual disks. 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. Virtual disks facilitate namespace distinctions of different views of a file, provide a solution to the caching problem, and also allow service binding at the device level.

Mounting a virtual disk creates virtual namespace for files on the device that the virtual disk is hooked to. A mount option allows users to specify which directory on the device should be exported through the virtual disk. Figure 1 shows an example of how virtual files and virtual disks relate to the base file and physical device. In the example, three virtual disks (`/dev/vdisk1`, `/dev/vdisk2` and `/dev/vdisk3`) are hooked to disk `/dev/disk1`. Mounting these virtual disks exports virtual views of all files descending from `/foo` (specified in the mount option) under `/vd1`, `/vd2` and `/vd3` respectively. The three virtual 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.

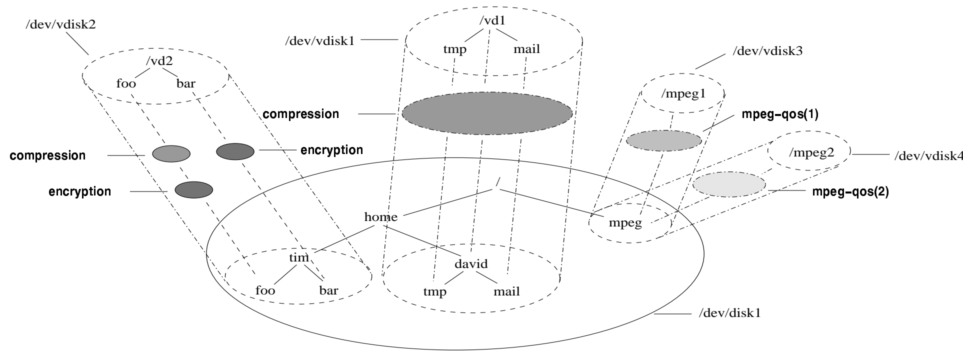


Figure 2. Example of service binding in MVSS

We developed the multi-view file system (MVFS) for management of virtual files in MVSS. MVFS is a stackable file system layer based on the vnode structure, it sits on top of the native file systems and forwards file system operations such as name resolution to them. Stackable file system structure is discussed in [16].

2.2 Application Interface

MVFS provides a flexible interface — *attach* on the host for users to associate services to virtual files. *Attach* has the following general interface: *attach(virtual file, service name, parameters)*. Examples of *parameters* include keys for encryption, query criteria for database SELECT operation, quality of service (QoS) parameters for MPEG filters, etc. Continuing with the previous example in Figure 1 and assuming that `/foo/bar` is an encrypted file, we can associate the decryption service with the virtual file `/vd1/bar` by issuing *attach(/vd1/bar, decryption, key)*. As a result, `/vd1/bar` now provides a decrypted (using the specified key) view of `/foo/bar`. The *attach* interface allows service binding to be separated from other file operations such as open, read and write. This separation has the advantage that accesses to the virtual file after the *attach* operation will automatically see the attached view, thus allowing transparent enhancements without modifying existing application codes.

The *attach* interface provided by MVFS supports service binding at the granularity of a single file. This is done for both flexibility and more efficient use of the virtual namespace. A user can apply different enhancements to different directories or files on the same virtual disk. Attaching service to a directory affects all the files and subdirectories under it. MVFS allows the user to decide whether the attached directory or file should inherit services attached to its ancestor directories.

Figure 2 shows an example of the service binding in MVSS. All the virtual disks are hooked to `/dev/disk1`. Virtual disk `/dev/vdisk1` is mounted on `/vd1` and exports `/home/david`. Attaching a compression service to `/vd1` allows all files under `/vd1` to be transparently compressed on disk. Virtual disk `/dev/vdisk2` is mounted on `/vd2` and exports `/home/tim`. File `/vd2/bar` is attached with an encryption service, whereas `/vd2/foo` is attached with 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` export `/mpeg` on `/mpeg1` and `/mpeg2` respectively. Both `/mpeg1` and `/mpeg2` are attached with an MPEG-transform service but with different quality parameters. Users accessing files under `/mpeg1` will see the MPEG files under `/mpeg` but at quality level 1, and under `/mpeg2` at quality level 2.

MVFS saves the service binding information (service name and the parameters) for a virtual file in a data structure that is associated with each virtual inode, called the AUX area [1]. The AUX area also contains information such as which virtual blocks on the virtual disk are allocated to the virtual file.

2.3 Device Model

We assume that the storage devices in MVSS have enough resources to generate the specified views of the data. These resources usually consist of a processor with sufficient amount of memory and a light-weight embedded OS. These requirements are quite reasonable for future disks.

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

The idea of using virtual block addresses is based on the following observation: capacities of block devices are usually much less than the maximum value that the operating system could support. For example, on a system that uses

[1] AUX stands for “auxiliary”.

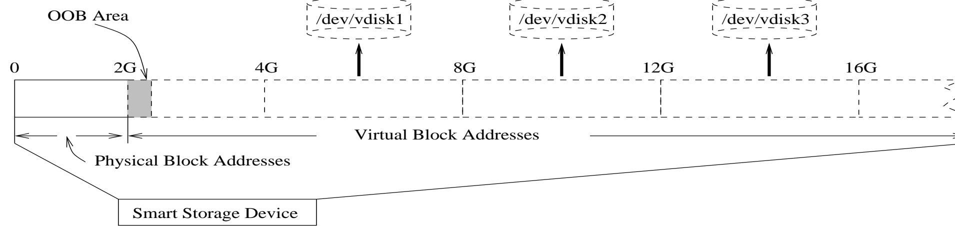


Figure 3. An example of virtual block address space management in MVSS

32-bit integers to represent block numbers, a block device could reference up to 2^{32} blocks, which means a capacity of 4TB with a block size of 1KB. Current disk capacities are much smaller than 4TB. We use the extra block addresses beyond the physical capacity of the disk as virtual block addresses. A similar approach has been recently adopted for building a flexible memory controller [9].

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

MVSS uses "out-of-band" communication between the host and the devices to maintain meta data on the devices. These meta data contain information that enables the devices to carry out the requested processing for active data requests. To support "out-of-band" communication, each physical device reserves a certain range of virtual block addresses, called the OOB area. In MVSS, the host sends messages to the device by writing data blocks into its OOB area in a way similar to memory-mapped IO.

Figure 3 shows an example of how MVSS manages a device's virtual block address space. The storage device in the example has a physical capacity of 2GB. Each virtual disk hooked to it is allocated 4GB of the device's virtual block space. IO requests forwarded from a virtual disk to the smart storage device contain only block numbers within its allocated virtual block space. This enables the device to find out from the address range of an IO request the virtual disk the request belongs to.

2.4 Service Binding at Device Level

The *attach* interface allows users to associate services with virtual files at the file level. The service binding information needs to be passed down to device level, where the services are performed. At the device level, the concept of files is not available. Instead of introducing new interfaces between the file system and the device, MVSS uses the existing operating system's interfaces.

Each virtual file in MVSS is allocated some virtual blocks when it is accessed for the first time. MVSS associates these virtual blocks to the AUX area of the virtual file through a virtual block map. Given a virtual block number, the map allows us to find the corresponding AUX area of the virtual file that the virtual block belongs to. There is one virtual block map for each hooked virtual disk in the system. MVSS replicates the virtual block maps and all the AUX areas of the virtual files in use on the corresponding storage device through "out-of-band" communication to the device. This enables the device to find out which AUX area is associated with the requested data based only on the virtual block addresses and to process the data according to the service binding information in the AUX area. MVSS uses the OOB area to maintain on the device up-to-date copies of the virtual block maps (one for virtual disk hooked to the device) and all the AUX areas in use.

A virtual block map containing binding information for every virtual block on a virtual disk would be inefficient and difficult to manage. The space requirement for such a map would also be prohibitive.

To solve this problem, MVSS uses a dynamic virtual block allocation scheme. We group virtual blocks into segments and divide the virtual block map into zones which contains segments of different sizes. Both segment sizes and number of zones in a virtual block map can be tailored according to the workload. 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 is accessed again. Cached blocks belonging to a segment need to be flushed from the buffer cache before the segment could be allocated to other virtual files. 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 file system 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 1GB disk only takes 8KB. Maps of this size could fit into the memory on the disk easily.

The use of virtual block addresses requires the system to

support file block offset to physical block number conversion (i.e., the *bmap* interface on traditional UNIX file systems) on the disk side. This requirement does not increase the complexity of our device model too much, as 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.

2.5 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 — filter applet onto the device. Filter applets contain codes to be invoked on the data that is being read from or written to the device. Applets could be in any format as long as the disk OS supports them. For example, they could be Java codes. Filter applets are similar to stream modules in that: filter applets get data from one end and produce output data on the other end; filter applets can be pipelined (i.e., the output of one filter applet serves as the input of another filter applet).

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

First, 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). IO requests from the host are passed to working processes through IPC mechanisms.

Second, filter applets can only access disk resources through a set of interfaces provided by the disk OS. To ensure proper sharing of resources such as CPU and memory, MVSS allows the administrator to classify applets into different classes and assign to each class an execution priority and the maximum amount of resources to be allocated.

Third, filter applets can only initiate IO requests in a limited way. Certain filter applets may need to initiate IO requests on their own. However, allowing a filter applet to access any blocks on the disk would break file system protection. In the Active Disk model [1], disklets are blocked from initiating IO requests and have to rely on host-resident codes to issue IO requests for them. MVSS allows filter applets to initiate IO requests, but only to those data blocks belonging to the base file of the virtual file containing the requested virtual blocks. In this way, mistakes or malicious codes in filter applets could not result in unauthorized access or harm the integrity of the file system and meta data on the disk. They can do no more damage than a normal user process that runs on the host accessing the same file.

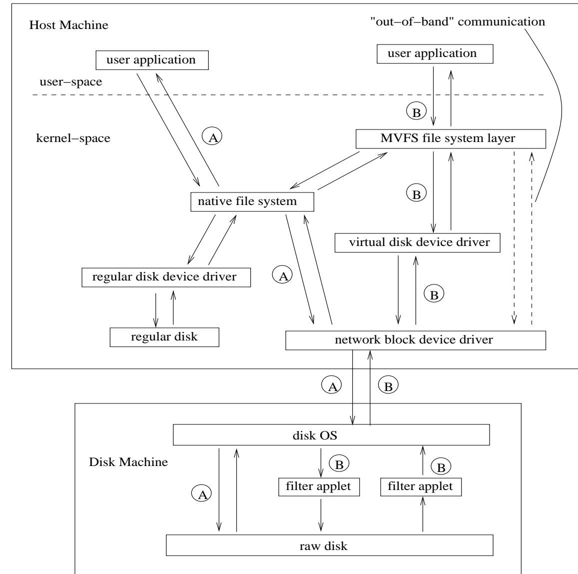


Figure 4. System structure of our MVSS prototype

3 Implementation and Results

In this section, we describe some details about our prototype implementation. We then present experiment results from several applets we developed.

3.1 System Configuration and Structure

Our MVSS prototype is built on two 166MHz PCs running Linux. One of them acts as the smart disk, another as the host. The host and the disk machine are interconnected through a dedicated 10Mbps Ethernet switch. The disk read throughput for the disk machine is about 7MB/s.

We simulate the smart disk by running a user space daemon on the disk machine. The daemon uses a 1G byte partition on the disk machine's SCSI disk as the raw disk, it performs IO operations directly through the device file. This is done to simulate a "smart disk" that may be available in the future.

We use a modified network block device (NBD) driver on the host to communicate with the disk machine. We implemented both MVFS and the virtual disk device driver as loadable kernel modules.

Figure 4 illustrates the system structure of our prototype implementation. The virtual disk device driver forwards IO requests to the NBD driver after we hook the virtual disk to the NBD. "Out-of-band" communication and IO requests for both NBD blocks and virtual disk blocks all go through the common block-level interface provided by the NBD driver. They are then demultiplexed on the disk

Table 1. Complexity of Example Applets

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

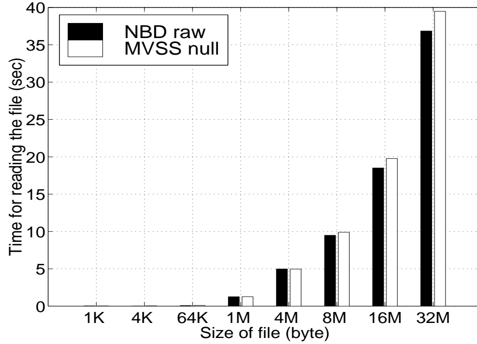


Figure 5. Overhead of MVSS with null applet

machine based on their block numbers and processed separately. Path A in the figure shows data flows for IO requests to normal files, path B shows those for IO requests to virtual files.

3.2 Results and Analysis

We developed the following filter applets: null, crypt, database selection and MPEG QoS filtering. The development of these applets shows that MVSS greatly reduces the development and maintenance effort. Table 1 gives the size of each applet in number of lines of C code. MVSS makes it practical for normal end users without knowledge of the file system internals to develop and apply device level enhancement according to their needs.

The following subsections discuss each applet in detail and present experiment results.

Null The null applet just copies the input data to the output buffer without any processing. We use the null applet to measure the overhead of our system.

Figure 5 shows the overhead of reading virtual files attached with the null applet in our prototype implementation. The results show that our implementation adds an overhead of about 8 percent for accessing virtual files compared to normal files. Currently we use a user-space daemon on the host to manage the AUX areas and virtual block allocation for easy of development. We expect to see a lower overhead after we move this code into kernel space. Another part of the overhead is caused by extra data copying.

Crypt Cryptographic techniques are playing an increasingly important role in modern computing system security, however, user-level tools are usually cumbersome.

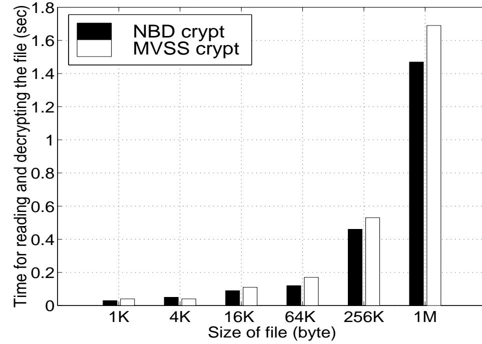


Figure 6. MVSS performance with crypt applet

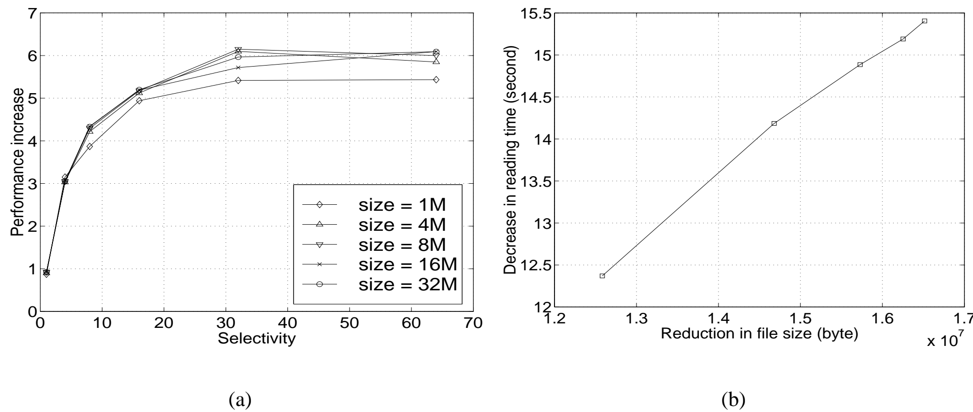
Adding cryptographic support at system level provides better transparency. Secure storage at system level can be achieved in MVSS by using a filter applet that encrypts the data blocks on writes and decrypts it on reads. Keys are specified during the attach operation as parameters to the crypt applet. Different keys can be used for separate files and directories.

We developed a crypt filter applet using the algorithm in the Unix `crypt` utility. In our implementation, we store the hashed key as applet parameter in the AUX area of the virtual file. Access to attached virtual files is controlled by restricting the virtual directories through the standard UNIX file protection mechanism.

Figure 6 shows the throughput of reading an encrypted file in MVSS with the crypt applet (MVSS crypt). It is compared with that of 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 percent for MVSS similar to what is observed with the null applet (again we expect this overhead to be reduced when MVFS is fully implemented in kernel space). Since the decryption work is now migrated to the disk, the host CPU usage for MVSS crypt is only 7 percent compared to 55 percent for NBD crypt. This reduced load will improve throughput in a system with multiple disks.

Database Selection Decision support and data warehousing database workloads comprise an increasing fraction of the database market today, and requirement for IO capacity and processing continues to grow rapidly. To meet this need, several researchers have proposed device models such as Active Disk and IDISK. These benefits can be achieved in MVSS by loading a database selection applet onto the device that filters the dataset and only returns selected records. Similar data-intensive applications have been discussed in [18, 11, 15].

We created datasets with 64-bytes long records and developed a simple filter applet to filter records based on a selectivity parameter passed to the applet. For example, a se-



(a) (b)

Figure 7. Results with simple database selection filter applet

Table 2. 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

lectivity factor of 8 returned 1/8th of the total dataset. This simulates non-index SELECT operations that require scanning the entire dataset. Figure 7(a) shows that MVSS gains a performance increase of about 3.5 fold at selectivity 4 and 7 fold at selectivity 64. In our system, the I/O interconnection throughput is the bottleneck compared to the disk IO bandwidth. However, as the selectivity increases, the demand on the interconnection bandwidth reduces, making the disk IO time the bottleneck. As a result, the speedup does not scale with higher selectivity (without a corresponding increase in disk bandwidth). Figure 7(b) shows that the time differences are proportional to the bandwidth reductions. The results show the benefits of MVSS over a traditional storage system when the IO interconnection between host and disks is the bottleneck.

MPEG It is suggested in [19, 5] 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.

We developed an MPEG filter applet to show how a smart disk can be turned into an MPEG-aware disk in our system. Table 2 shows the results obtained by applying an MPEG filter applet on the video streams in MPEG files to throw away video data for B and P frames. We have also implemented a finer-level MPEG filter applet that discards slices of picture frames instead of entire frames. In Table 2, we show the throughput of reading an MPEG file in the following cases: 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.

These experiments demonstrate the advantages of MVSS over traditional storage systems. Results from the crypt and MPEG applets show that for computation intensive services, migrating processing to the devices leads to significant reduction in CPU load on the host machine. This allows MVSS to exploit the parallelism of providing services on multiple active storage devices concurrently. Results from the database selection and MPEG applets also show that MVSS can increase throughput by reducing the data traffic between the host and the devices. We plan to evaluate our system with more disk machines in future experiments.

Our current prototype implementation is read-only. MVSS is currently being enhanced to provide support for writes.

4 Related Work

In response to the increasing storage and computational demand for applications such as decision support database, multimedia, the Active Disk and IDisk models [1, 15, 11] 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. An evaluation of the active disk model for decision support database is provided in [18] for active disks against two alternative architectures: shared memory multiprocessors (SMPs) and workstation clusters. MVSS draws much inspiration from these work. Our work focuses on a real implementation and how to exploit the benefits of active storage devices within the existing file systems. MVSS supports a block-level interface unlike the stream model proposed in earlier approaches. Also the scheduling and mixed workload issues are not addressed in the earlier work.

The derived virtual device (DVD) model [14] proposed in the Netstation project provides a mechanism for safe shared device access in an untrusted environment by creating DVDs and managing them through a network virtual device manager. The proposed third-party transfer scheme using DVDs is similar to that in NASD. The Linux NBD that is used in our prototype is similar to their virtual Internet SCSI adapter [13].

Virtual disks [2] and logical disks [4] have been proposed to improve storage organizations and file systems. Virtual disks in MVSS are a different generalized abstraction of a storage device.

Stackable file system allows extension of functionalities for existing file systems through Vnode Stacking [16, 17, 7], which allows the interposition and composition of vnodes so that file system modules could be layered on top of each other. Earlier work on stackable file system has been focused on file level enhancement and does not support service migration to the devices.

5 Conclusions

We have proposed a storage system that allows flexible service migration to the storage devices. 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 onto the device. Results from a Linux PC-based prototype system demonstrated the effectiveness of MVSS.

We plan to port more services onto MVSS to demonstrate the flexibility of the approach.

References

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

[2] C. R. Atanasio et al. Design and implementation of a recoverable virtual shared disk. *IBM Tech. Report: RC 19843*, Nov. 1994.

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

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

[5] 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.

[6] G. A. Gibson et al. File server scaling with network-attached secure disks. In *ACM Sigmetrics*, June 1997.

[7] J. S. Heidemann and G. J. Popek. File system development with stackable layers. In *Trans. on Comp. Sys.*, 1994.

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

[9] J. B. Carter et al. Impulse: Building a smart memory controller. *Proc. of 5th Int. Symp. on HPCA*, Jan. 1999.

[10] J. Hartman et al. Joust: A platform for liquid software. *IEEE Computer*, 1999.

[11] 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.

[12] G. Ma and A. L. N. Reddy. An evaluation of storage systems based on network-attached disks. *Proc. of ICPP*, Aug. 1998.

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

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

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

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

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

[18] M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. *Proc. of HPCA*, Jan. 2000.

[19] A. Vahdat, T. Anderson, and M. Dahlin. Active names: Programmable location and transport of wide-area resources. In *USENIX Symposium on Internet Technologies and Systems*, 1999.