MVSS: an Active Storage Architecture

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. A number of studies have

demonstrated the benefits of migrating services to different levels in the system [1, 2, 3, 4]. These studies have motivated the utility of devices with more intelligence and function. Device level enhancements such as compression, encryption, and log-based file systems [5] are being proposed. Direct network-attachment of disks is being proposed to enable data transfers from the device directly to the client rather than through the server [6, 7, 8, 9, 10]. Earlier work on active disks [11, 12, 13] proposes to migrate part of the application to be executed at the device resulting in "filtered" data. There are two main advantages of active disks. One is the parallelism available among the disks. There could be more aggregate CPU power at the disks than at the server. The other is the ability to dramatically reduce bandwidth demands on the I/O interconnect by filtering data at the disks. In many systems today, interconnect bandwidth is often a significant bottleneck.

Previous work has shown that active disks can significantly improve the performance and reduce the cost of a system. However, one of the most difficult problems in implementing active storage systems is how to migrate services onto storage devices. 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 [4], 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 usually large part of the OS 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 ex-

isting 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.

- 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. Section 4 discusses the results of running several applications on the prototype. In Section 5, we compare various aspects of MVSS with related work. Section 6 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 kept the following principles in mind 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.

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 (we will discuss this in section 2.4).

Mounting a virtual disk creates the virtual namespace for files on the device that the virtual

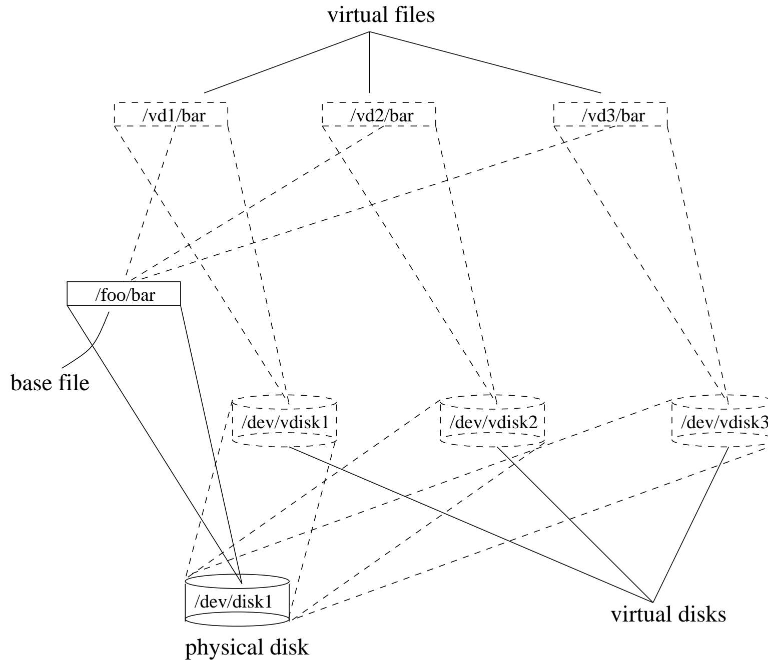


Figure 1: Concept of virtual files in MVSS

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.

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 [14]. 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 [15].

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

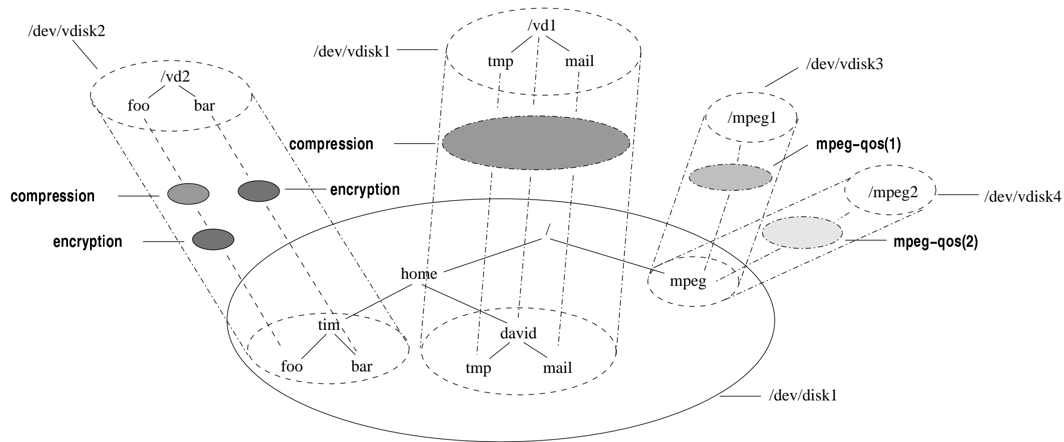


Figure 2: Example of service binding in MVSS

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

[1] AUX stands for "auxiliary".

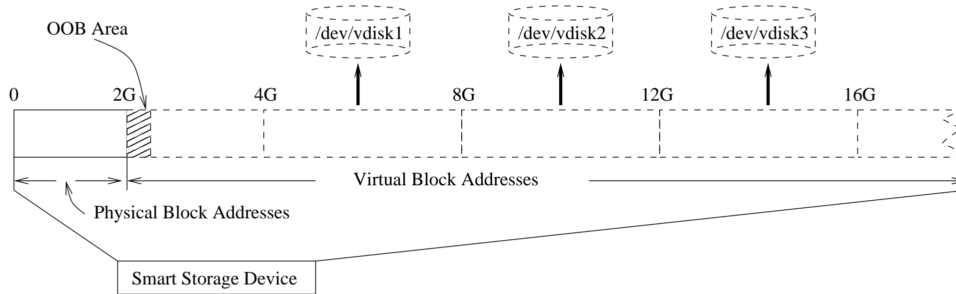


Figure 3: An example of virtual block address space management in MVSS

support. For example, on a system that uses 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. A disk's virtual block space can be expanded through the use of larger block size or 64-bit block addresses. 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 [16].

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 the virtual block space

allocated to that virtual disk. 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 the 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. The device can then 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 each 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. Fortunately, in our system, it is not necessary to use such fine-grained virtual block maps for managing virtual blocks. Instead, 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). When a virtual file is accessed for the first time, the system allocates virtual blocks

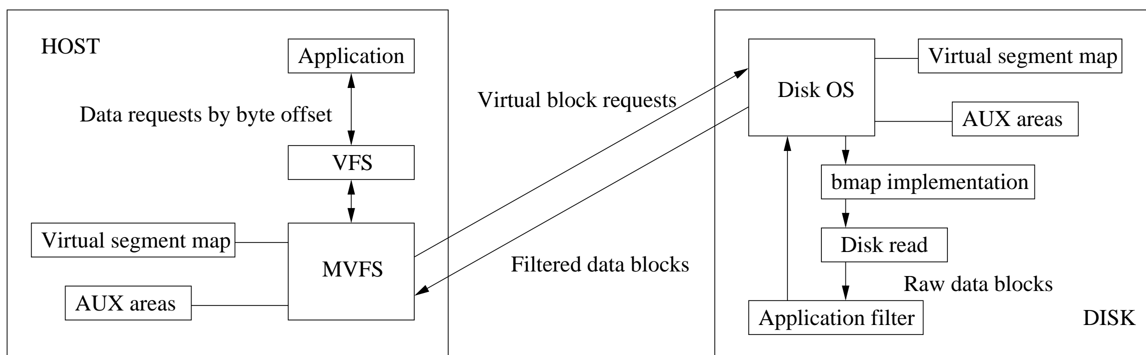


Figure 4: Service binding at device level in MVSS

to it one segment at a time, starting from the zone that contains segments of the smallest size. Our 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 (the smallest segment size is set to 1000 blocks). Maps of this size could fit into the memory on the disk easily.

The disadvantage of such a coarse-grained virtual block allocation scheme is that virtual block numbers may be wasted due to internal fragmentation. This is not a problem in our system for the following two reasons. First, since most systems only allocate cache buffers when the data are accessed, the internal fragmentation caused by the use of large segments will not result in any actual system resources being wasted. Second, virtual blocks 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. There is no need to reserve the virtual block numbers allocated to a virtual file forever, because cached blocks of that virtual file will be purged out of memory gradually if not accessed. The number of virtual files that are frequently accessed on the system is normally limited during any period of moderate length. It is possible that a virtual file may be allocated some segments that were allocated to another virtual file previously while some of the data blocks in those segments still exist in cache. In this case, these cache buffers need to be invalidated before being accessed. To make maximum use of the system buffer cache, MVSS always tries to allocate the same virtual segments to a virtual file when it is accessed again.

Based on the segment map, the disk can find out which AUX area the requested virtual block is associated quickly. It can further derive the block offset of the requested block in the virtual file since the AUX area contains information about all the virtual segments allocated to the virtual file. In the simplest case where there is an one-to-one mapping between the virtual blocks and the data blocks in the base file, the disk just needs to feed the corresponding data at the same block offset in the base file to the application filter. For more sophisticated mapping, the application code is responsible for deciding which data blocks need to be read from the base file to generate the requested data block. In both cases, the block offsets in the base file need to be translated into physical block numbers (which is handled by the `bmap` function in traditional UNIX file systems). There are two ways of doing this. One approach is to let the disk query the host for the translation. The disadvantage is the extra overhead and latency caused by doing this. The other is to port the `bmap` function on the disk, since usually all the meta data required for the translation are already on the disk. Here we assume that there are no dirty meta data on the host (write support and cache coherency will be discussed in section 2.6). We take the second approach in our system. This does not increase the complexity of our disk model too much, as our prototype implementation shows that it takes only around 100 lines of C code to provide complete Linux ext2 file system `bmap` support on the disk. Figure 4 illustrates how our scheme works.

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 [11], 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.

2.6 Write Support and Cache Coherency

The procedure for writing to a virtual file is symmetric to that of reading a virtual file and is straightforward to implement, when the operation does not change the size of the base file. Otherwise, block allocation and deallocation are required. Implementing these functionalities on the disk is more complicated than porting the `bmap` function. A preallocation scheme such as those in parallel file systems [17] can be used.

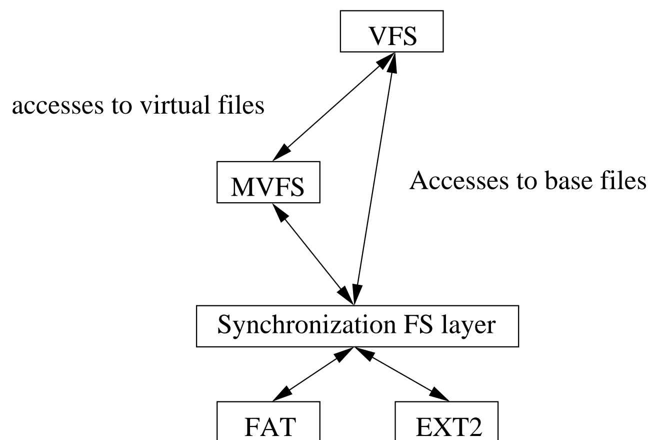


Figure 5: Write support and cache coherency through the synchronization layer

So far in our discussion we assume that all the data and meta data on the disk are up-to-date. This may not be true if the base file is modified on the host. Dirty data buffers on the host may cause filter applets to return out-of-date results, while dirty meta data may cause the disk OS to let filter applets access wrong blocks. Writes to base files also raise the cache coherency problem: it is possible that out-of-date virtual data blocks exist in the cache after the base file is modified. Some applications may require all data blocks belonging to a base file and its virtual files in the buffer cache to be synchronized. A general framework for cache coherency at block granularity is difficult to build since the mapping between virtual blocks and the corresponding base blocks is application-dependent and may change dynamically during run-time. In [18], the authors proposed a coherency scheme for stackable file systems that attacks a similar problem. Their approach, however, uses a custom cache manager and cache-object naming scheme. The approach also requires each service layer to participate in cache-object management to provide service-specific mapping information between cache objects at different layers. To apply such a scheme in MVSS would incur severe performance penalty because the services are provided on the disks while the cache manager resides on the host. Also, such a strict synchronization scheme may not be necessary for all applications. For example, video files on a network video server are rarely modified.

Our solution to these problems is to use a light-weight file-level consistency and coherency

scheme. Compared to the object-level cache coherency scheme in [18], our approach is less efficient but much simpler to implement. Figure 5 shows how the scheme works. The basic idea is to add a synchronization layer on the host to coordinate all accesses to a base file and its virtual files. The synchronization layer itself is just a stackable file system that sits directly on top of the native file systems but below MVFS. Each vnode in the synchronization layer contains the necessary state information and acts like a centralized synchronization point. For example, the synchronization layer will mark a base file as "dirty" after it is modified. If any of its virtual files is read later, the synchronization layer will invalidate existing cache buffers for that virtual file, and call `fsync` upon the base file to flush all dirty data and meta data to disk before allowing the read to continue. The synchronization layer also provides a central file lock which is used to prevent possible race conditions between modifications to the base file and accesses to its virtual files. The synchronization layer adds very little overhead to file operations since it does not introduce extra data copying. It's important that applications do not access base files directly through the native file systems. This can be done through mount point overlay or by hiding the mount point of the underlying native file systems from user applications.

2.7 Miscellaneous issues

Disks are random-access storage devices. However, applications may require data blocks of a file to be processed in certain order. For example, some compression or encryption algorithms require the data to be decompressed or decrypted sequentially. Restricting the access patterns of such a virtual file will not be enough. If the virtual file is accessed by several processes simultaneously or part of the data are already in the buffer cache, the data blocks may still be requested out-of-order at device level. Our system does not pose any restriction on how a virtual file is accessed, it's the responsibility of the corresponding filter applet to ensure that correct filtered data are returned.

When volume manager (VM) or RAID is used, data may be stripped across several disks.

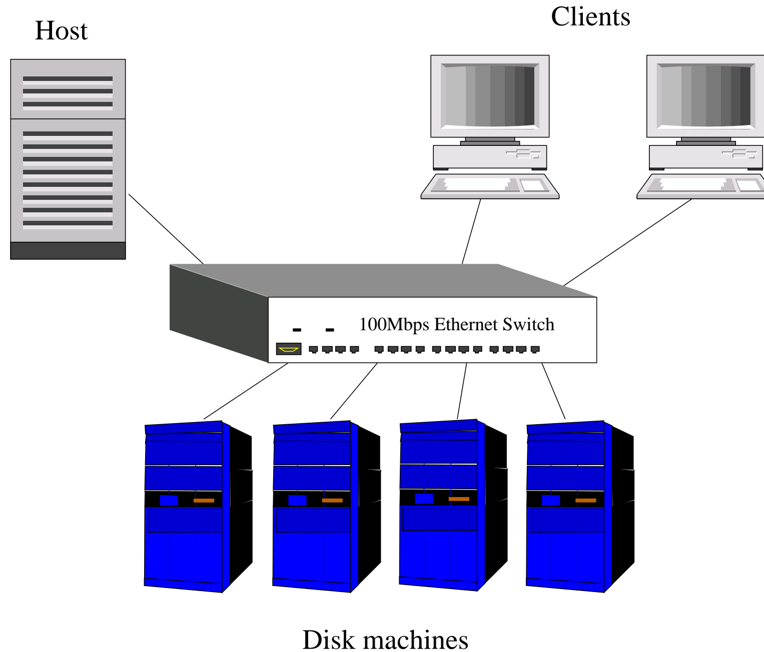


Figure 6: Testbed structure

This is not a problem if the filter applets are executed on the RAID controller or if the data are organized in such a way that the filter applet can process each strip of data without accessing data blocks or other disks. Otherwise, inter-disk communication is required. A VM also adds another level of indirection between file system and disk data layout. With VM, the information returned by `bmap` may no longer be enough. If the VM is implemented at the disk side (on the disk controller), then porting `bmap` to disk is sufficient since the volume to device mapping will be done on the disks. If the VM is on the server, it may need to be modified to provide support for resolving the physical block numbers on disks in MVSS.

3 Implementation

In this section, we describe some details about our prototype implementation. The structure of our testbed is shown in figure 6. Our test configuration includes a 233MHz pentium PC as the host and a few 166MHz pentium PCs as the smart disks. Each machine has 64MB of memory, a 100Mbps network interface card and is connected to each other through a dedicated 100Mbps Ethernet switch. The average disk read throughput on the smart disks is about 7MB/s.

We use the Linux Network Block Device (NBD) driver on the host to communicate with the smart disks. NBD was developed to allow one machine to use files or disks on another remote machine as its local block devices through TCP/IP. An NBD server runs as a user space daemon on each disk machine. During the initialization of an NBD, an NBD client process is started on the host. The NBD client establishes a TCP connection with the NBD server and requests information such as the exported device size. The NBD client then associates the TCP socket with the NBD and sets the related device parameters in the kernel. After that, it then makes an *ioctl* call into the kernel and waits for data replies from the NBD server. In the current NBD implementation, there is one NBD client process for each initialized NBD device.

When a user application process tries to read data from an NBD device, the file system resolves the block addresses and passes the requests down to the NBD device driver. The NBD device driver then sends the requests to the NBD server through the TCP socket associated with the device. After receiving the requests, the NBD server reads the data from the disk and sends it back through the TCP connection. When the replies arrive at the host, the NBD client process wakes up and puts the reply data into the corresponding cache buffers (the NBD client is not involved when data requests are sent to the NBD server). The user process is then woken up to copy the data from buffer cache to the user space and the read is finished. The procedure for writing data to an NBD is similar.

Figure 7 shows the structure of our system (the synchronization file system layer is omitted to avoid clogging the figure). Solid lines illustrate the structure and data path of a traditional system with NBD. Dashed lines shows the new components and data path in MVSS. On the host side, we modified the NBD device driver slightly to support virtual block requests. We implemented both MVFS and the virtual disk driver (VD) as loadable kernel modules. MVFS manages the AUX areas for virtual files and the segment maps for virtual disks. File system operations such as name resolution are forwarded to native file systems (e.g., EXT2). The virtual disk driver forwards I/O requests to the underlying device driver. After a virtual disk is hooked to an NBD, "Out-of-band" communication and I/O requests for both NBD blocks and

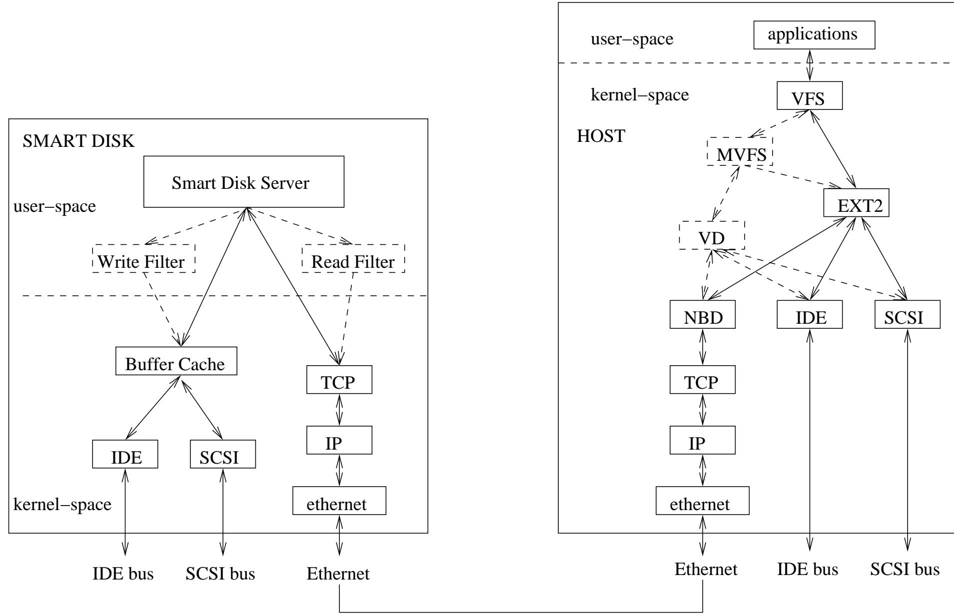


Figure 7: MVSS structure

virtual disk blocks all go through the common block-level interface provided by the NBD driver. They are then demultiplexed on the disk machine based on their block numbers and processed separately. On the disk side, we run our smart disk server instead of the original NBD server. The smart disk server is a multi-threaded user-space process. One of the threads simulates a simplified disk OS. Other threads simulate working processes. The OS thread receives all the requests from the host and dispatches them to the other threads. We only show the data path on the smart disk side for requests sent through Ethernet in the figure. Data paths for active I/O requests from the SCSI or IDE interface are similar.

We used the Linux kernel network QoS support [19] to simulate different link speeds between the host and the devices. The kernel QoS support provides a framework for controlling and rate limiting bandwidth for specified network flows. With this facility, the link bandwidth between the disks and the host was varied from a minimum of 10 Mbps to a maximum of 100 Mbps. Figure 8 shows NBD throughput for sequential reads on one of the smart disks with different interconnect bandwidth between the host and the smart disk. Figure 9 shows the overhead of reading a virtual file under MVSS when no filter is attached. The results show that our MVSS prototype implementation only adds a small overhead of about 3-4% over NBD performance.

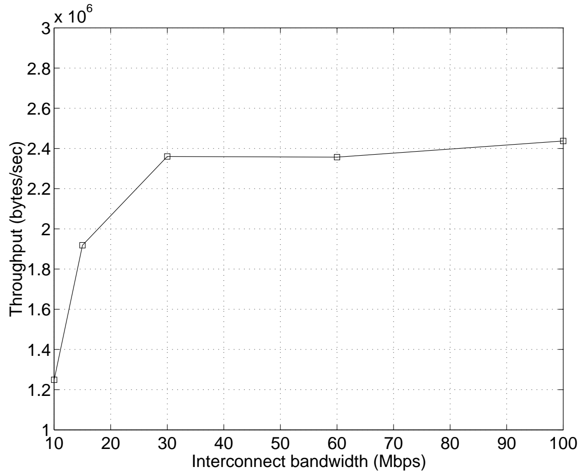


Figure 8: Single NBD disk throughput under different link speed

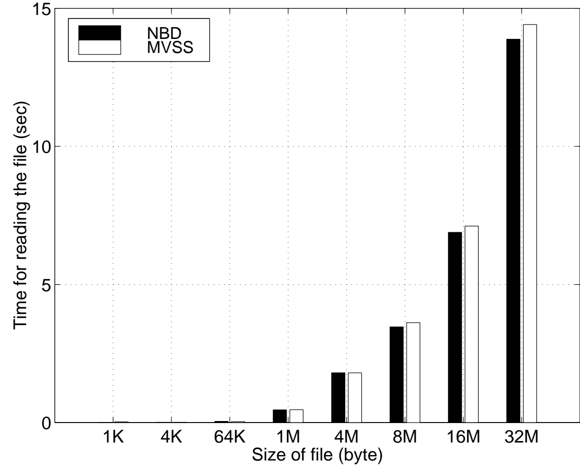


Figure 9: Overhead of MVSS

In our experiments, we let the smart disk servers on the disk machines directly operate on the disk device file to eliminate file system overhead. However, data still go through the system buffer cache as Linux currently does not support raw device interface. Because of the multiple data copies and context switches, the maximum throughput achieved by a single NBD in our system is limited to about 2.4 MB/s. Moving the NBD server process into kernel space on the disk machine will improve the throughput by avoiding extra data copying. Other optimizations for implementing block level storage over IP are discussed in [10]. The absolute performance numbers in our measurements are low by today's standards due to the age of the equipment used, but the relative numbers and conclusions remain valid.

4 Applications and Results

To study the benefits of active storage, we chose a few real-world applications: encryption, MPEG QoS filtering and image processing with median filter. Encryption shows the advantage of moving computation to the disks. The encryption algorithm does not change the data set size. MPEG QoS filtering is an example application that benefits from both parallelism and data traffic reduction from active disks. Image processing with median filter is an example application where the data processing work can be split between the host and the disks. In this

Table 1: Complexity of Example Applets

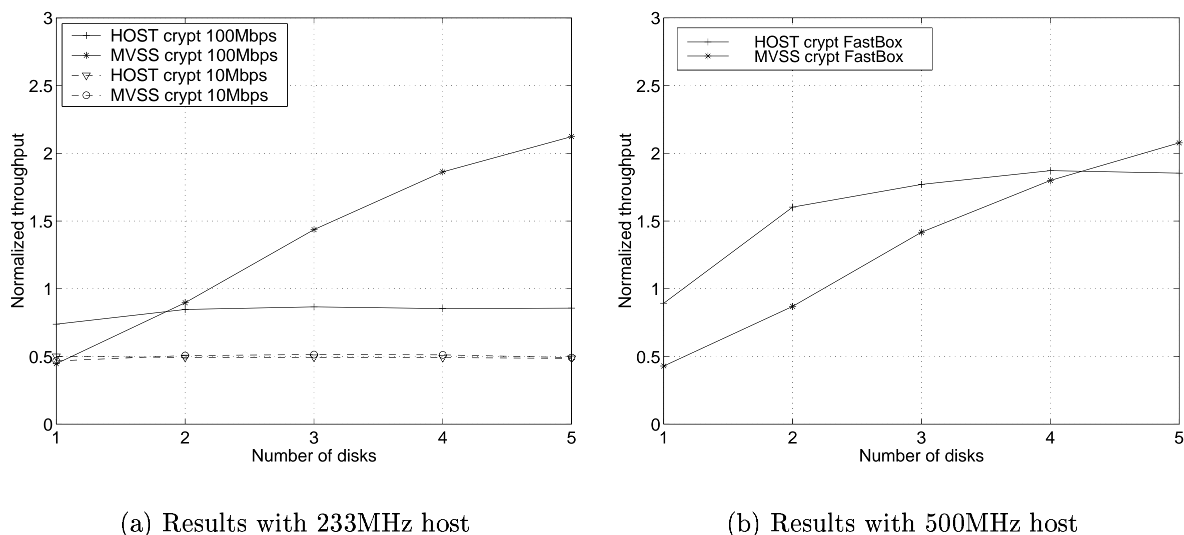
Filter Applet	Lines of C codes
Blowfish Encryption	250
MPEG QoS	220
Median filter	160

section we discuss each applet in detail and present experimental results.

We developed filter applets for each application. Table 1 gives the size of each applet in lines of C code. Developing applets in our system is similar to developing normal user space application and requires no knowledge of the file system internals. In our system, it is practical for normal end users to develop and apply device level enhancements according to their needs.

Application 1: Blowfish Crypt Cryptographic techniques are becoming increasingly important in modern computing system security. However, user-level tools are usually cumbersome. Adding cryptographic support at system level provides better transparency [20]. With the current trends in storage pooling and outsourcing, data are increasingly being stored encrypted on the device. Secure storage at system level can be achieved in our system by using a filter applet that encrypts the data blocks on writes and decrypts them 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 Blowfish algorithm [21]. We store the hashed key as an applet parameter in the AUX area of the virtual file. Access to the attached virtual files is controlled by restricting the virtual directories through the standard UNIX file protection mechanism.

Figure 10(a) shows the total throughput of reading encrypted files attached with the blowfish decryption filter (MVSS crypt) with different number of disks. The results are compared with those of reading the files through NBD and decrypting them on the host inside a user process(HOST crypt). The data were measured by running several processes on the host at the same time, each process reading a file from one of the disks. We decrease the amount of data read from each disk as the number of total disks is increased so that the total amount of



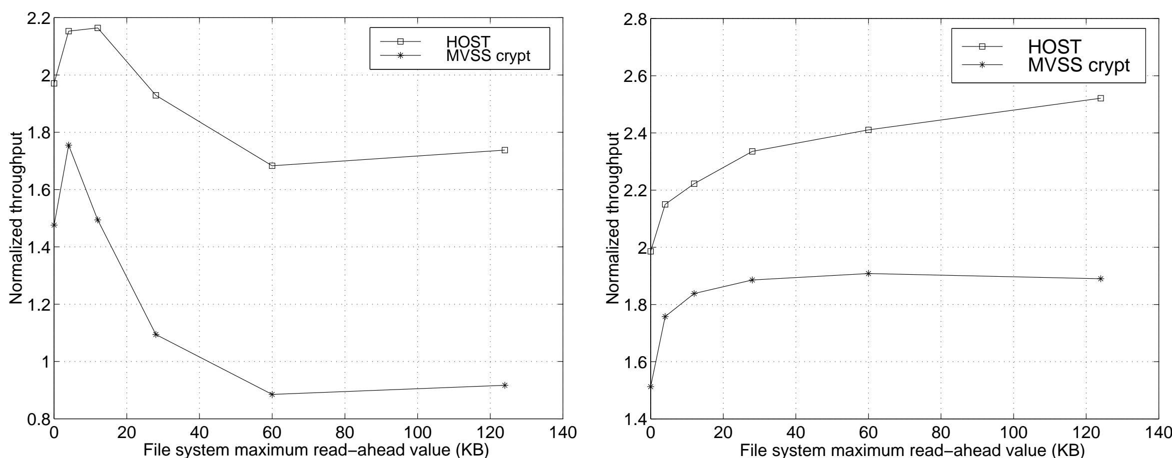
(a) Results with 233MHz host

(b) Results with 500MHz host

Figure 10: Performance with the Blowfish encryption applet

data read remains constant. The results are normalized with the read throughput of a single NBD disk with 100Mbps interconnect bandwidth. The results show that, with one disk, the throughput of HOST crypt is higher than that of MVSS crypt because the host processor is more powerful than the disk processor. As the number of disks increases, the throughput of HOST crypt is limited to 2.1 MB/s as the host CPU is saturated. For MVSS crypt, since the processing is now moved to the disk, the host CPU is no longer the bottleneck. The result shows that the total throughput increases almost linearly as the number of disks increases. With concurrent I/O from five disks, MVSS crypt is able to achieve a speed up of about 250% over HOST crypt. This shows the advantage of MVSS when decrypting multiple files from multiple disks concurrently (alternately, when encrypted files are stripped across multiple disks).

Figure 10(a) also shows the results when the interconnect bandwidth was set to 10Mbps. The results for HOST crypt and MVSS crypt are almost identical. Both are limited by the low interconnect bandwidth. The results show that the ability to exploit the CPU power on the disks could be severely limited when the interconnect bandwidth is low. While some applications (as we show later) may benefit from the reduction of data movement across the I/O interconnect fabric, other applications may continue to require high bandwidth I/O interconnects.



(a) Request queue of length 128 (b) Request queue of length 2048

Figure 11: Performance with different file system read-ahead values

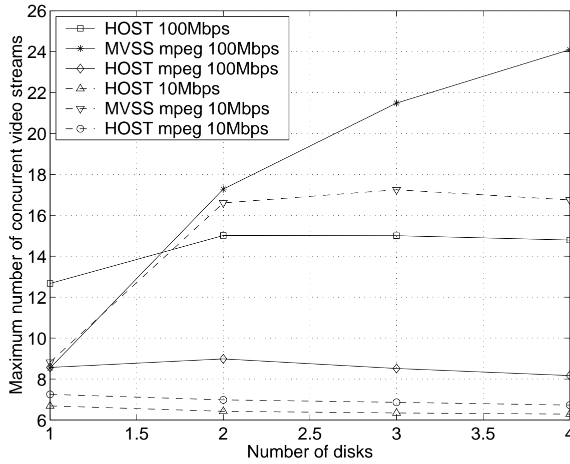
Figure 10(b) gives the results when we replaced the 233MHz host with a faster machine, which has a 500MHz processor and 128MB of memory. The results show that the faster host improves the throughput of HOST crypt. MVSS crypt does not see significant differences in performance since most of the processing work is done at the disks, which remain unchanged. Now, the active storage configuration does not achieve a higher throughput until the system employs five disks. This experiment shows that if the server machine is considerably more powerful than the disks, performance gains with active storage may not be apparent unless significant number of disks are employed in the system. It is to be noted again that the performance of the encryption application is primarily dependent on processing power and does not see any gains through reductions in data traffic across the I/O interconnect fabric.

During our experiments with multiple disks, we noticed that large file system read-ahead (or prefetch) values resulted in less total throughput for the system than small read-ahead values. Further investigation showed that the reason for this is that Linux uses a common fixed length request queue for requests to different block devices. Since the average delay of NBD replies is significantly longer than those of local disk I/O requests, large file system maximum read-ahead value allows one process to use up all the available requests (even when the read

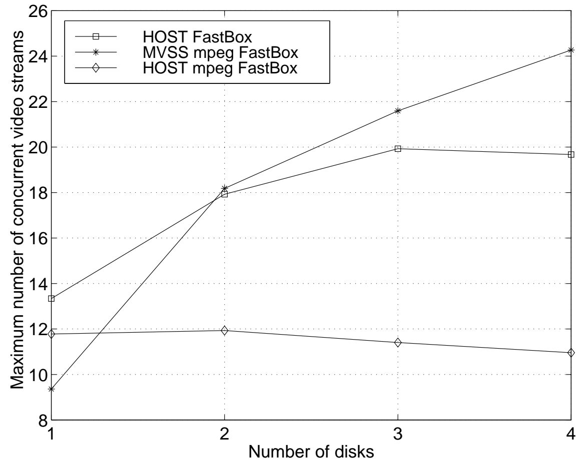
process is reading one data block at a time), forcing other reading processes to wait before their requests could be sent to the devices. Virtual block requests asking for filtered data from the disk will experience longer delays due to processing at the disk. The limited queue length renders the host file system prefetching scheme less effective, reduces the parallelism between the disks and thus the total throughput. Figure 11(a) shows the total throughput for four disks with (MVSS crypt) and without (HOST) the encryption filter under different file system read-ahead values. Again the results here are normalized as in figure 10(a). The results show that with the system default queue length of 128 requests, the maximum total throughput is reached with a maximum read-ahead value of 12KB for HOST, and 4KB for MVSS crypt (where the encryption work on the disk introduces more latency for data replies). The default maximum read-ahead value is 124KB, at which point the MVSS crypt throughput drops to around 2.2 MB/s (about 90% of the NBD throughput for a single disk machine). The problem can be solved by increasing the length of the request queue or by employing an adaptive queue management scheme. Figure 11(b) shows the result after we increased the length of the request queue to 2048 requests.

Application 2: MPEG Many video player applications try to adapt the video data based on the characteristics of the client's connection, bandwidth, packet drop rates, etc. 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 our system, 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 and colour mode 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. The filter throws away video data for B and P frames in MPEG video streams. The size of a virtual file attached with the filter is reduced to about 1/3



(a) Results with 233MHz host



(b) Results with 500MHz host

Figure 12: Performance with MPEG application

of the original video file size for the MPEG video clip used in our experiment. We have also implemented a finer-level MPEG filter applet that discards slices of picture frames instead of entire frames. In our test, we simulate a network video server by running a user space NFS server over UDP on the host. We use another PC as the NFS client to simulate video clients issuing video requests by running a number of processes. Each process fetches one video file from one of the disks through the NFS server. We modified the NFS server to add MPEG filtering support. Figure 12(a) shows the results as the maximum number of video streams the system can support when the filtering is done on the disks (MVSS mpeg) and on the host by the NFS server (HOST mpeg). The figure also shows the results when no filtering is done on the host and the disks (HOST). We did the test using various number of disks with host-disk interconnect bandwidths of 10Mbps and 100Mbps. The results show that MVSS mpeg performs better than HOST mpeg for both low and high interconnect bandwidths. Even with a single disk, MVSS mpeg outperforms HOST mpeg in the low interconnect bandwidth (10Mbps) case due to the benefits of reduction in data movement across the I/O interconnection fabric. With larger number of disks, the increased aggregate processing power at the disks improves the performance further. We see that MVSS mpeg achieves a 3-fold speed-up compared to HOST

mpeg and a performance improvement of 60% compared to HOST with four disks.

With low interconnect bandwidth (10Mbps), HOST mpeg performance is better than that of HOST since less data is sent after filtering from the server to the clients. However, when sufficient network bandwidth (100Mbps) is available, HOST performs best for the single disk case since the MPEG filtering causes the CPU to be the bottleneck for both MVSS mpeg and HOST mpeg. With more than two disks, MVSS mpeg performs better than HOST (even MVSS mpeg 10Mbps outperforms HOST 100Mbps) as the performance of the latter is limited by the large amount of data the server needs to transfer between the disks and the client.

Again, figure 12(b) gives the results when we replaced the 233MHz host with the 500MHz faster machine. The faster server improves the throughput of HOST mpeg. But the active storage system still provides higher throughput with two or more disks. Compared to earlier results of the encryption application, the MVSS system performs better than the HOST system with the mpeg application. This is due to the fact that the mpeg filter applet exploits both the advantages of the active storage system: processing parallelism and I/O bandwidth reduction.

Application 3: Median Filtering In previous experiments, the data processing work is done entirely either on the host or on the disks. Applications whose work can be partitioned to run on the disks and the host at the same time may achieve better overall system utilization. Median filter based image processing is one example of such applications. Median filter is a non-linear filter that is widely used in image processing to remove shot-noise and random noise. A morphological median filter on a 3 by 3 kernel needs to find the median of 9 values for each set of 9 neighbor pixels in the input image. Figure 13 shows one way of splitting the work between the host and the disks. The image is partitioned into sub-images and stripped across the disks. The filter on each disk processes part of each sub-image in the center and sends the filtered data to the host. A user application running on the host reads the partially filtered sub-images and applies the same median filter on the unfiltered areas. The percentage of the work done at disk is $(M * M)/(N * N)$. By varying the value of M relative to N, we can vary the amount of work done at the disks and the host.

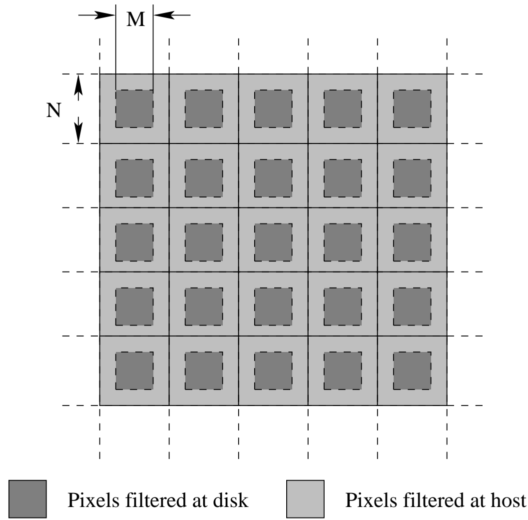


Figure 13: Median filter partition

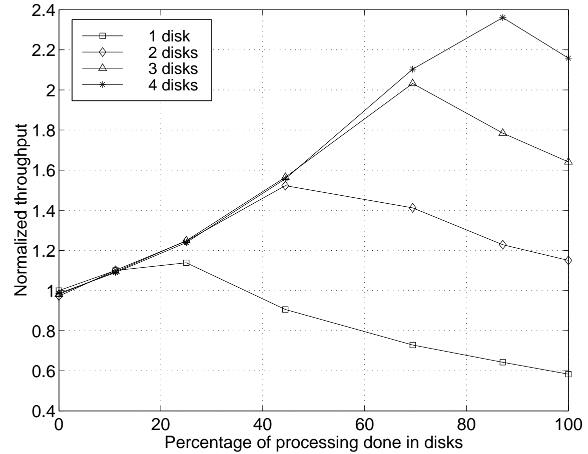


Figure 14: Performance with the medianfilter

In this experiment, we developed a simple median filter using a 3 by 3 kernel. The filter takes a parameter which allows us to specify the size (i.e., M) of the disk filtered area of each sub-image. We formatted the input image file so that the sub-image size is equal to the block size for convenience. Figure 14 shows the results obtained by varying M. The results are normalized with the performance of doing all the filtering on the host with only one disk. The result shows that the percentage of work done on the disks when the system achieves the most speedup increases as the aggregate computation power increases with the addition of more smart disks into the system.

Since most file systems automatically issue read-ahead requests, the advantage of parallelism between disks and host can be exploited by a normal single thread application (as the user application we ran on the host in this test). However, user applications can not change the file system read-ahead behavior according to their need, which may limit the potential parallelism achieved. On the other hand, in our system, users can develop applets that do prefetch on the devices according to application specific requirements.

Summary of application results

Our experience with the evaluation of these applications on the prototype can be summarized as:

- The design of our prototype system allowed the realization of active storage systems without significant modifications to file systems, host-disk interface and applications.

- The advantage of parallelism between active disks and the host can be exploited by a single-thread application through file system read-ahead mechanism.

- Active storage systems benefit applications through reduced data movement across the I/O interconnect fabric and increased parallelism in processing data at multiple disks. Applications that do not allow data traffic reductions may see increased latencies if parallelism through multiple disks cannot be harnessed (for example, due to limited interconnect bandwidth) because disk processors are normally less powerful than the host processor.

5 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 [11, 12, 13] 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 [22] 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.

The derived virtual device (DVD) model [23] 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 [10].

Virtual disks [24] and logical disks [25] 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 [14, 26, 15], 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.

6 Conclusions

We have proposed an active storage system that allows flexible migration of application level code to the storage devices. The multi-view storage system, MVSS, provides multiple views of an underlying file, similar to the multi-view database systems providing multiple views of the underlying database. We have shown that it is possible to retain the block-level interface of the devices while allowing flexible service deployment within the existing file systems without significant changes to the OS. We also showed that it is possible to build intelligent devices without porting significant amounts of file system functionality onto the device. Results from a Linux PC-based prototype system demonstrated the effectiveness of our approach. We also investigated many architectural issues such as the impact of system level parameters including the interconnect bandwidth, number of devices, and relative processing power of the devices and the server.

References

[1] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting. Liquid software: A new paradigm for networked systems. Technical report, Department of Computer Science, The University of Arizona, 1996.

[2] 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.

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

[4] 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*, April 1999.

[5] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual log based file systems for a programmable disk. *Proc. of 3rd Symp. on Operating Systems Design and Implementation*, Feb. 1999.

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

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

[8] 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.

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

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

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

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

[13] 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.

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

[15] John Heidemann and Gerald Popek. File system development with stackable layers. In *Transactions on Computing Systems*, 1994.

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

[17] Steve Cannon. Concurrent file system - making highly parallel mass storage transparent. *Proc. Supercomputing*, April-May 1989.

[18] John Heidemann and Gerald Popek. Performance of cache coherence in stackable filing. In *Proceedings of the 15th Symposium on Operating Systems Principles*, December 1995.

[19] Werner Almesberger, Alexey Kuznetsov, and Jamal Hadi Salim. Differentiated services on linux. In *Proceedings of Globecom'99*, pages 831–836, December 1999.

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

[21] B. Schneier. *Applied Cryptography*, chapter Blowfish, pages 336–9. John Wiley & Sons, second edition edition, 1996.

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

[23] 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.

[24] 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.

[25] 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.

[26] G. C. Skinner and T. K. Wong. "stacking" vnodes: A progress report. In *USENIX Conference Proceedings*, pages 161–174, 1993.