# MVSS: An Active Storage Architecture

Xiaonan Ma, *Member*, *IEEE Computer Society*, and
A.L. Narasimha Reddy, *Senior Member*, *IEEE Computer Society*

**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 multiview 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 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 example applications implemented on the prototype and discuss a variety of architectural issues including mixed workloads.

**Index Terms**—Active storage, virtual disks, service migration, multiple views, virtual block addresses.

## 1 Introduction

Many researchers have suggested migrating services to storage devices to improve overall system performance. A number of studies have demonstrated the benefits of migrating services to different levels in a system [1], [2], [3], [4]. These studies have motivated the utility of devices with more intelligence and function. Direct network-attachment of disks is being proposed to enable data transfers from devices directly to clients rather than through servers [5], [6], [7], [8]. Earlier work on active disks [9], [10], [11] proposed to migrate parts of applications to storage devices resulting in "filtered" data. There are two main advantages of active disks. One is the parallelism available among disks. There could be more aggregate CPU power at disks than at servers. The other is the ability to dramatically reduce bandwidth demands on I/O interconnects by filtering data at 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 issues in implementing active storage systems is how to migrate services onto storage devices. In traditional storage systems, storage devices normally provide a block-based interface. File systems access data on storage devices through block addresses. The problem is that flexible service migration to 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], a 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 major components of operating systems, 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 multiview 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 a file are provided to users through the 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 generic block 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 operating systems.
- 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 and Section 5 discuss the results of running several applications on the prototype and the results of mixed workloads. In Section 6, we compare various aspects

- *X. Ma is with IBM Almaden Research Center, 650 Harry Road, K56/B3, San Jose, CA 95120-6099. E-mail: xiaonma@almaden.ibm.com.*
- *A.L. Narasimha Reddy is with the Department of Electrical Engineering, Texas A&M University, 214 Zachry, College Station, TX 77843-3128. E-mail: reddy@ee.tamu.edu.*

*Manuscript received 14 Sept. 2001; revised 21 Apr. 2003; accepted 24 Apr. 2003.*
*For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 114981.*

of MVSS with related works. Section 7 concludes the paper and points to future work.

## 2 Design of MVSS

The major design objective of MVSS is to enable the migration of services to storage devices within existing file systems. We kept the following principles in mind while designing MVSS: 1) keep the software layer on disks 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 a file and certain services. Virtual disks are place holders for virtual files. Second, MVSS provides a flexible application 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 generic block-based interface is employed in MVSS. Fourth, MVSS makes service binding information for each virtual file available at the device level through virtual block addresses.

These key ideas in MVSS work together to provide two interfaces. The idea of virtual files and the application interface provide the high-level interface between the user domain and operating systems, which makes different service configurations available through the file system namespace, while virtual block addresses provide the low-level interface between operating system and storage devices for service migration to disks. In the following sections, we describe these ideas in detail.

### 2.1 Virtual Files and Virtual disks

In MVSS, a virtual file provides a view of an underlying file (called base file). 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 question: If 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, 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 discuss this in Section 2.4).

Mounting a virtual disk creates the 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. Fig. 1 shows an example of how virtual files and virtual disks relate to base files and underlying physical devices. 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, but do not have any physical data blocks associated with them.

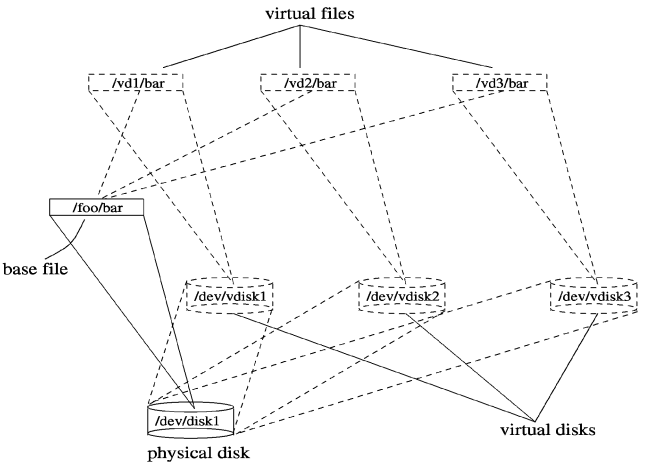



Fig. 1. Concept of virtual files in MVSS.

We developed the multiview file system (MVFS) for the management of virtual files in MVSS. MVFS is a stackable file system layer based on the vnode structure [12]. 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 [13].

### 2.2 Application Interface

MVFS provides a flexible interface—*attach* on hosts for users to associate services to virtual files. *Attach* has the following 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 Fig. 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 users to decide

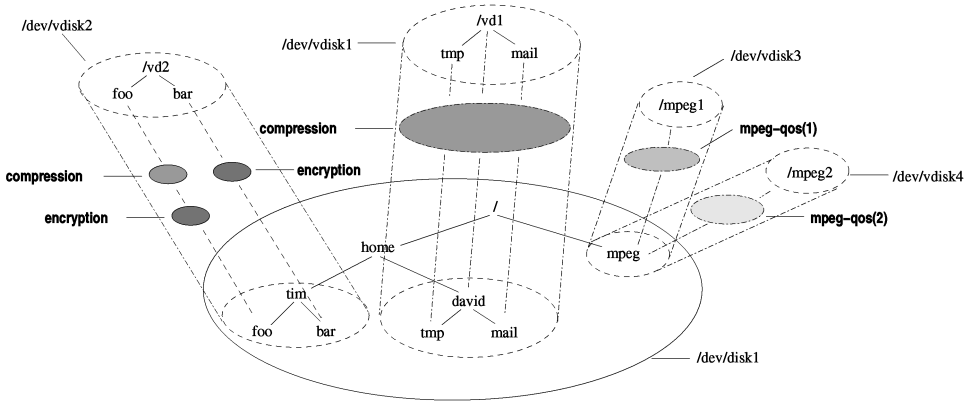



Fig. 2. Example of service binding in MVSS.

whether the attached directory or file should inherit services attached to its ancestor directories.

Fig. 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 the 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 compressed first and then encrypted as they are written, and the procedure is reversed when they are read. Virtual disks `/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 parameters) for a virtual file in a data structure that is associated with each virtual inode, called AUX area.[1] An AUX area also contains information such as which virtual blocks on the corresponding virtual disk are allocated to the virtual file.

## 2.3 Device Model

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

Devices in MVSS use the same block-based interfaces 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 a 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 values that operating systems could 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 4 TB with a block size of 1 KB. Current disk capacities are much smaller than 4 TB. A disk's virtual block space can also 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 a disk as virtual block addresses. A similar approach has been recently adopted for building a flexible memory controller [14].

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 those virtual disks hooked to the same device do not overlap with each other. Virtual block addresses allocated to a virtual disk will be allocated to the virtual files on that virtual disk.

MVSS uses "out-of-band" communication between hosts and storage devices to maintain meta data on devices. These meta data contain information that enables 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 OOB area. In MVSS, a host sends messages to a device by writing data blocks into its OOB area in a way similar to memory-mapped IO.

Fig. 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 2 GB. Each virtual disk hooked to it is allocated 4 GB of the device's virtual block space (the size of virtual block space for each virtual disk may be different from each other and can be dynamically adjusted at run time). 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 services are performed. At the device level, the concept of files is not available. Instead of introducing new interfaces between file

1. AUX stands for "auxiliary."

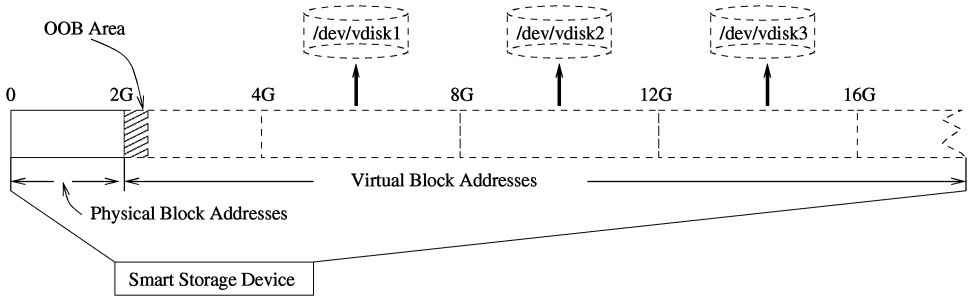



Fig. 3. An example of virtual block address space management in MVSS.

systems and devices, MVSS uses existing operating systems' 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 each 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 maintains on the corresponding storage 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 through "out-of-band" communication using the OOB area. This enables the device to find out which AUX area is associated with the requested data based only on the virtual block addresses of a request. The device can then process the data according to the service binding information in the AUX area.

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. 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 a virtual block map into zones which contain segments of different sizes (both segment sizes and number of zones in a virtual block map can be tailored according to the workload of an virtual disk). When a virtual file is accessed for the first time, the system allocates virtual blocks to it one segment at a time, starting from the zone that contains segments of the smallest size. Our virtual block allocation scheme allows service 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 size of the segment map for a 1 GB disk was set to 32 KB (the smallest segment size is set to 1,000 blocks). Maps of this size could fit into the memory on disks 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 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 a virtual file 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 in a 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 the new data can be accessed. To make maximum use of the system buffer cache, MVSS always tries its best to reallocate the same virtual segments to a virtual file.

Based on virtual block maps, a disk can find out which AUX area a requested virtual block is associated with 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 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 are discussed in Section 2.6). We took the second approach in our system. This does not increase the complexity of our disk software layer 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. Fig. 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 devices. Filter applets contain codes to be invoked on the data that is being read from or written to a device. Applets could be in any format as long as the disk operating system supports them.

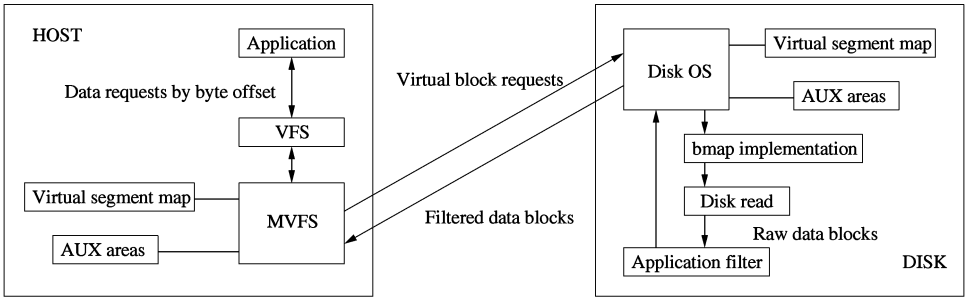



Fig. 4. Service binding at device level in MVSS.

For example, they could be Java codes. Recent work has also shown that it is possible to use scripting languages such as Tcl for developing secure active services [15]. Filter applets are similar to stream modules in that: filter applets get input 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 to another filter applet).

MVSS allows filter applets to be loaded onto the devices by normal users. Since filter applets are executed at the device level, without protection, a mistake in the codes 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 the process level on disks (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 operating system. To ensure proper sharing of resources such as CPU and memory, MVSS allows administrators 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 a disk would break file system protection. In the Active Disk model [9], 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 accesses or harm the integrity of the file system and meta data on a disk. They can do no more damage than a normal user process that runs on hosts 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 storage devices is more complicated than porting the `bmap` function. A preallocation scheme, such as those in parallel file systems [16], can be used.

So far, in our discussion, we assume that all the data and meta data on disks are up-to-date. This may not be true if base files are modified on hosts. Dirty data buffers on hosts may cause filter applets to return out-of-date results, while dirty meta data may cause disk operating systems 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 a 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 the 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 runtime. In [17], Heidemann and Popek 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 a significant performance penalty, because services are provided on disks while cache managers reside on hosts. 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 [17], our approach is less efficient, but much simpler to implement. The basic idea is to add a synchronization layer on hosts 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 is important that applications do not access

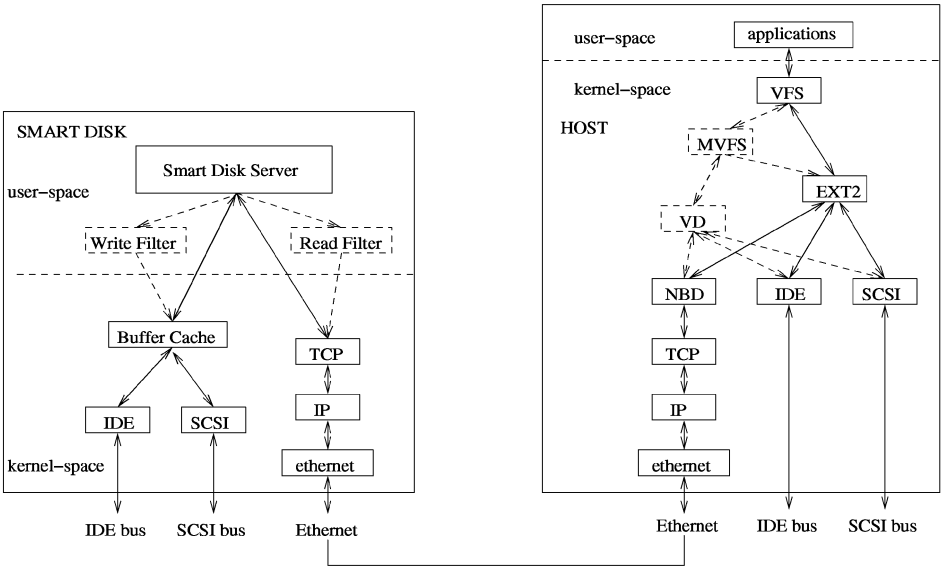



Fig. 5. MVSS structure.

base files directly through the native file systems. This can be done through mount point overlay or by hiding the mount points 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 a certain order. For example, some compression or encryption algorithms require 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 virtual file is already in the buffer cache, the data blocks may still be requested out-of-order at the device level. Our system does not pose any restriction on how a virtual file is accessed; it is the responsibility of the corresponding filter applet to ensure that correct filtered data are returned.

When a volume manager (VM) or a RAID system is used, data may be striped across several disks. This is not a problem if the filter applets are executed at the RAID controller or if data are organized in such a way that filter applets can process each strip of data without accessing data blocks or other disks. Otherwise, interdisk communication is required. A VM also adds another level of indirection between file systems and disk data layout. With a VM, the information returned by `bmap` may no longer be enough. If the VM is implemented at the disk side (on disk controllers), then porting `bmap` to disk is sufficient since the volume to device mapping will be done on disks. If the VM is on hosts, 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. Our test configuration includes a 233 MHz pentium PC as the host and a few 166 MHz pentium PCs as the smart disks. Each machine has 64 MB of memory, a 100 Mbps network interface card, and is connected to each other through a dedicated 100 Mbps Ethernet switch. The average disk read throughput of the smart disks is about 7 MB/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 configuration 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.

When a user application process tries to read data from an NBD, the file system resolves the block addresses and passes the read 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.

Fig. 5 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 NBDs. 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

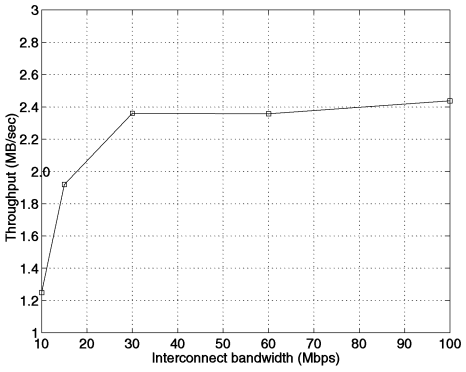



Fig. 6. Single NBD disk throughput under different link speed.

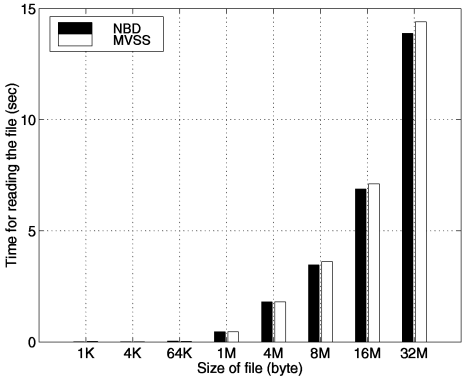



Fig. 7. Overhead of MVSS.

manages AUX areas for virtual files and virtual block 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 virtual disk blocks all go through the block-based interface provided by the NBD device 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 operating system. Other threads simulate working processes. The operating system thread receives all the requests from the host and dispatches them to 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 using the SCSI or IDE interface are similar.

We used the Linux kernel network QoS support [18] 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. Fig. 6 shows NBD throughput for sequential reads on one of the smart disks with different interconnect bandwidth between the host and the smart disk. Fig. 7 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 percent over NBD performance.

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 got through the system buffer cache (the Linux raw device support was not stable at the time of the experiments). 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 the 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 [8]. The absolute performance numbers in our measurements are low by today's standards due to the age of the equipments 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 a median filter. Encryption shows the advantage of moving computation to disks. The encryption algorithm does not change the size of data. MPEG QoS filtering is an example application that benefits from both parallelism and data traffic reduction. Image processing with a median filter is an example of applications where the data processing work can be split between hosts and disks. In this section, we discuss each applet in details and present experimental results.

We developed filter applets for each application. Developing applets in MVSS is similar to developing normal user space applications and requires no knowledge of file system internals.

### 4.1 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 [19], [20]. With the current trends in storage pooling and outsourcing, data are increasingly being stored encrypted on devices. Secure storage on devices can be achieved in MVSS by using a filter applet that encrypts data blocks on writes and decrypts them on reads. Keys are specified during *attach* operations 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]. Hashed keys are stored as applet parameters in the AUX areas of virtual files. Accesses to decrypted virtual files are controlled by restricting the virtual directories through the standard UNIX file protection mechanism.

Fig. 8a shows the total throughput of reading encrypted files through virtual files attached with the blowfish decryption filter (MVSS crypt) with different numbers of disks. The results are compared with those of reading the encrypted files through NBD and then decrypting them on

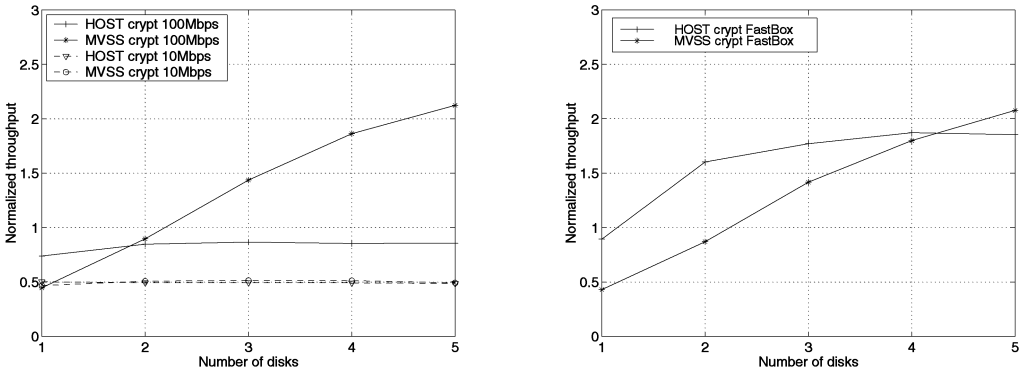



Fig. 8. Performance with the Blowfish encryption applet. (a) Results with 233 MHz Host and (b) results with 500 MHz host.

the host inside a user process (HOST crypt). The throughput was 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 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 became 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 percent over HOST crypt. This shows the advantage of MVSS when decrypting multiple files from multiple disks concurrently (or when encrypted files are striped across multiple disks).

Fig. 8 also shows the results when the interconnect bandwidth was set to 10 Mbps. 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.

Fig. 8b gives the results when we replaced the 233 MHz host with a faster machine, which has a 500 MHz processor and 128 MB 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 hosts are considerably more powerful than disks, performance gains with active storage may not be apparent unless significant number of disks are employed in the system.

During our experiments with multiple disks, we noticed that large file system read-ahead 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 shared 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 values allow one process to use up all the available request entries in the queue (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. Fig. 9a 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 Fig. 8a. The results show that with the default queue length of 128 requests, the maximum total throughput is reached with a maximum read-ahead value of 12 KB for HOST, and 4 KB for MVSS crypt (where the encryption work on the disk introduces more latency for data replies). The default maximum read-ahead value is 124 KB, at which point the throughput of MVSS crypt drops to around 2.2 MB/s (about 90 percent 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. Fig. 9b shows the result after we increased the length of the request queue to 2,048 requests.

### 4.2 MPEG QoS Filtering

Many video player applications try to adapt video data based on the characteristics of clients, such as connection bandwidth, packet drop rates, etc. For example, it makes little sense to send a high quality color MPEG stream to a hand-held device with a small black and white screen behind a low-bandwidth wireless link. MPEG files are large

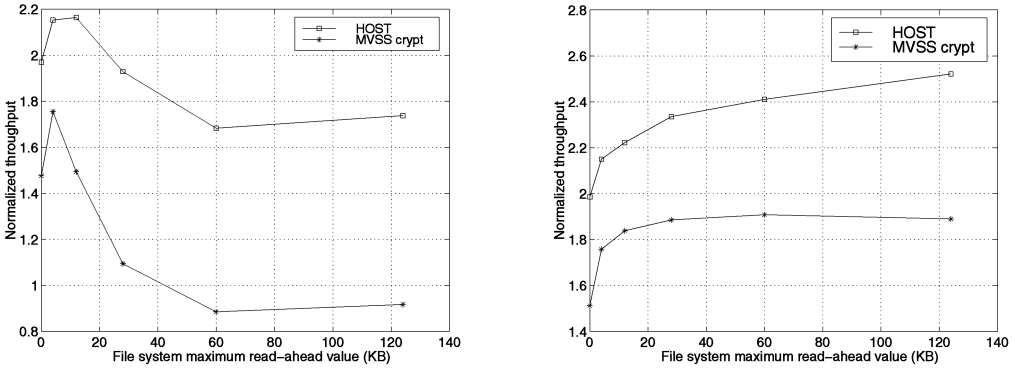



Fig. 9. Performance with different file system read-ahead values. (a) Request queue of length 128 and (b) request queue of length 2,048.

enough that storing the same video file at different levels of quality on disks 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, and color mode during 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 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 the experiment, we simulated a network video server by running a user space NFS server over UDP on the host. We used 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 also modified the NFS server to add MPEG filtering support. Fig. 10a shows the maximum numbers 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 different numbers of disks with the host-disk interconnect bandwidth set to 10 Mbps and 100 Mbps, respectively. 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 (10 Mbps) case due to the benefit of reducing data traffic 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 three-fold speed-up compared to HOST mpeg and a performance improvement of 60 percent compared to HOST with four disks.

With low interconnect bandwidth (10 Mbps), 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 (100 Mbps) 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 10 Mbps outperforms HOST 100

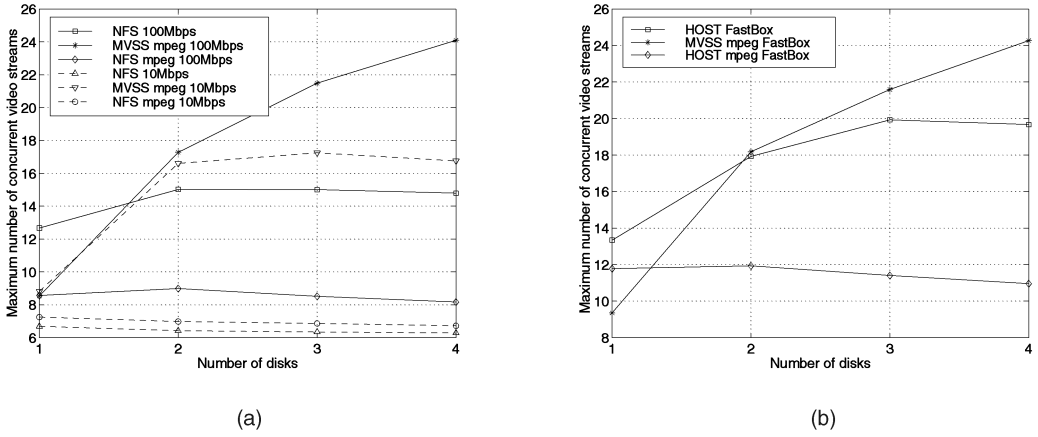



Fig. 10. Performance with MPEG application. (a) Results with 233 MHz host and (b) results with 500 MHz host.

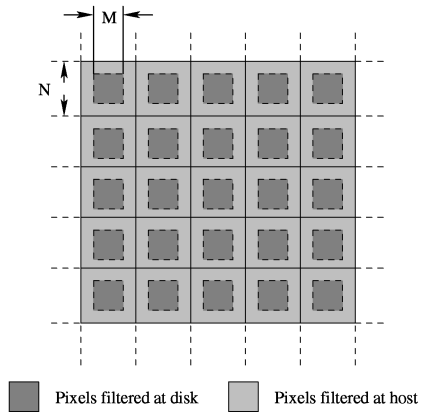



Fig. 11. Median filter partition.

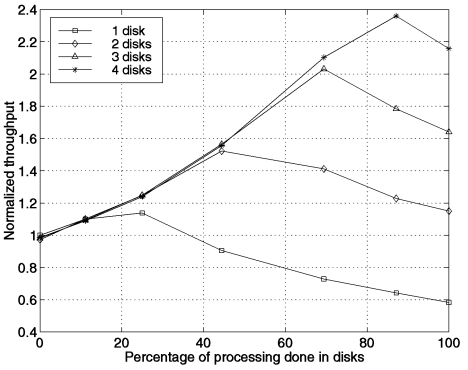



Fig. 12. Performance with the median filter.

Mbps) 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, Fig. 10b gives the results when we replaced the 233 MHz host with the 500 MHz 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 relative to the HOST system with the mpeg application. This is due to the fact that the mpeg filter applet exploits both the advantages of active storage systems: processing parallelism and I/O bandwidth reduction.

### 4.3 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. Image processing with a median filter is one example of such applications. Median filter is a nonlinear filter that is widely used in image processing to remove shot-noise and random noise. A morphological median filter on a $3 \times 3$ kernel needs to find the median of nine values for each set of nine neighbor pixels in the input image. Fig. 11 shows one way of splitting the work between the host and the disks. The image is partitioned into subimages and striped across the disks. The filter on each disk processes part of each subimage in the center and sends the filtered data to the host. A user application running on the host reads the partially filtered subimages 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.

In this experiment, we developed a simple median filter using a $3 \times 3$ kernel. The filter takes a parameter which allows us to specify the size (i.e., M) of the disk filtered area of each subimage. We formatted the input image file so that the subimage size is equal to the block size for convenience. Fig. 12 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 maximum 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 hosts can be exploited by a normal single thread application (as the user application we ran on the host in this experiment). However, user applications cannot change the file system read-ahead behavior according to their need, which may limit the potential parallelism achieved. On the other hand, in MVSS, users can develop applets that prefetch data on disks according to application specific requirements.

### 4.4 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 interfaces, and applications.
- The advantage of parallelism between active disks and hosts can be exploited by a single-thread application through file system read-ahead mechanisms.
- 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 host processors.

## 5 Mixed Workload

To evaluate the performance of our system under mixed workloads, we created a workload of random I/Os along with active data requests from a database SELECT application. We generated datasets with 64 byte long records and developed a simple filter applet to filter the records based on a selectivity parameter passed to the applet. For example, a selectivity factor of eight causes the filter to return 1/8 of the total dataset. This simulates nonindex SELECT operations that require scanning entire datasets.

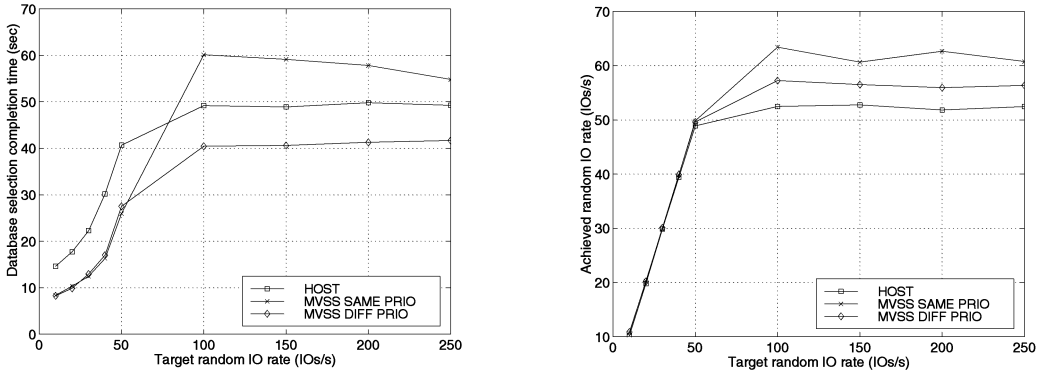



Fig. 13. Mixed workload with two threads given different priorities (100 Mbps interconnect BW). (a) Database select completion time and (b) achieved random I/O rate.

Random I/O requests are generated to retrieve data blocks at random locations on the disk. These I/O requests are regular read requests of 4KB each. In the experiment, we let 10 request processes issue random requests at varying rates. The rate is controlled by adjusting the time interval between the receipt of the data for the current request and the issuing of the next one. We varied the total target random I/O rate from five to 250 I/Os per second. The achieved random I/O rate may be lower than the target rate when the target request rate is beyond what the disk can handle.

We considered the case when two separate threads were used for active and normal requests, respectively, on the disk. In the first setting, we gave both threads identical priorities. In the second setting, we gave the active thread higher priority over the normal thread. We compare the results against those of the normal storage system, where the entire database file is transfered to the host and the database SELECT operation is carried out at the host.

Results in Fig. 13a give the time taken by the database SELECT application to finish processing a 32 MB data set at different target random I/O rates (the database application simply issues requests as fast as possible). Fig. 13b shows the actual number of random I/Os completed. We did the test with a selectivity of eight for the database applet and 100 Mbps interconnect bandwidth. When the active request processing thread is given a higher priority over the normal request processing thread (MVSS DIFF PRIO), the database application completes faster than when the threads are given equal priorities (MVSS SAME PRIO), while more random I/Os are supported in the latter case. When compared to the normal storage system (HOST), for MVSS DIFF PRIO, the database application finishes faster and higher random I/O rates are achieved at the same time. However, for MVSS SAME PRIO, the database application actually finishes slower than the normal storage system (HOST) when the target random I/O rate is beyond 100 IOs/sec. Because an active request requires more processing and I/Os than a normal request, by assigning the same priority to both threads, we implicitly penalized the active requests by allowing more random data requests to be processed (MVSS SAME PRIO achieves the highest rates for random data requests).

To understand the implications of the number of threads handling requests at the disk, we conducted another experiment with a total of five working threads on a single disk. The active data requests and random data requests were distributed across these five threads in various combinations. In one combination (5(A+R)), each thread obtains the next request from a common FIFO queue, whether it is an active request or a normal request, upon the completion of the thread's previous request. In other combinations, threads were dedicated to either handling active requests or handling random requests. For example, in 3A+2R, three threads handled active requests and two threads handled random requests. The results of these various combinations are compared with a normal system (5 NORMAL), where the database SELECT operation is carried out at the host. In this case, all requests arriving at the disk are normal requests and are handled by all the five threads through a common FIFO queue. We limited the interconnect bandwidth to 10 Mbps in this experiment. Fig. 14a shows the throughput of the database application, while Fig. 14b shows the achieved random I/O rates. As can be seen from the results, the performance of the active storage system is significantly impacted by the way the requests are distributed among the threads at the disk. When no thread is dedicated to handling random I/Os (5(A+R)), the threads spend most time on active requests. This results in considerably higher database SELECT throughput at the cost of low achieved random I/O rates. With four active request threads and one random request thread (4A+1R), the active storage system achieves higher database SELECT throughput and higher random I/O rate at all random I/O rates when compared with the normal system (5 NORMAL). In other cases, where the number of active request threads is close to or less than that of normal request threads, the database SELECT throughput of the active storage system becomes lower than that of the normal system (5 NORMAL) when the target random rate is beyond 100 IOs/sec. Again, this is because in the active storage system, the relative priority of random I/O request is implicitly increased through using dedicated random

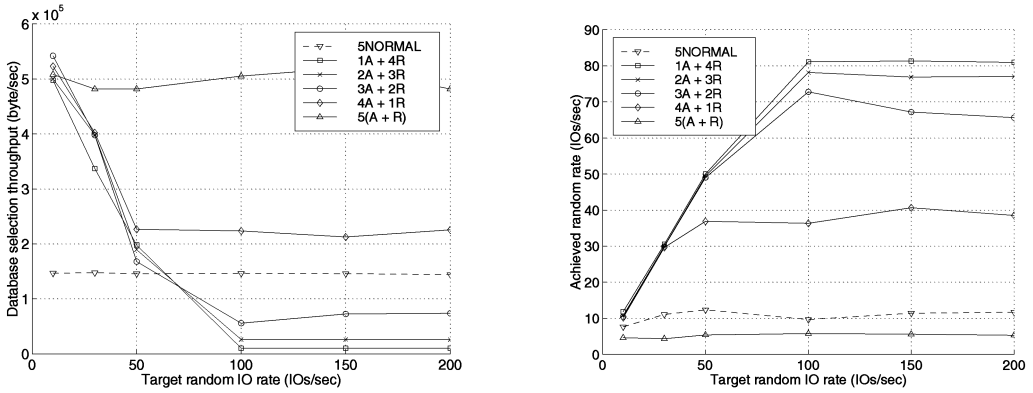



Fig. 14. Mixed workload with a total of five threads (10 Mbps interconnect BW). (a) Database select throughput and (b) achieved random I/O rate.

request threads, since the random request rate is much lower than the active request rate in the experiments. Compared with those in Fig. 13, the results here also show that the decrease in interconnect bandwidth from 100 Mbps to 10 Mbps had little impact on the active storage system, but significantly reduced the throughput of the normal system.

These results show the importance of request scheduling in an active storage system under mixed workloads. If some of the normal I/O requests are of interactive nature, it may be necessary to provide at least one dedicated thread at the disk to handle interactive requests in such workloads. The results also show that, sometimes, appropriate relative priorities need to be assigned to active and normal requests to achieve both the advantages of active storage and higher normal I/O throughput at the same time. It is an interesting subject for future research work to explore more sophisticated scheduling algorithms on storage devices to optimize the performance of different applications.

## 6 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 [9], [10], [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 [22]. An evaluation of the active disk model for decision support database is provided in [23] for active disks against two alternative architectures: shared memory multiprocessors (SMPs) and workstation clusters. MVSS draws much inspiration from these works. Our work focuses on a real implementation and how to exploit the benefits of active storage devices within existing file systems. MVSS supports a block-level interface unlike the stream model proposed in earlier approaches.

The derived virtual device (DVD) model [24] 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 [8].

Virtual disks [25] and logical disks [26] have been proposed to improve storage organizations and file systems. In Petal [27], a collection of storage servers present a unified, fault-tolerant virtual disk space to clients at block level. Virtual disk in MVSS is a different generalized abstraction of storage devices.

Stackable file system allows extension of functionalities for existing file systems through Vnode Stacking [12], [28], [13], which allows the interposition and composition of vnodes so that file system modules could be layered on top of each other. FiST [29] is a high level language that can generate codes for stackable file systems. Earlier work on stackable file system has been focused on file level enhancement and does not support service migration to devices.

The Semantic File System [30] uses semantic file indexing to provide an effective storage abstraction by mapping database queries onto a hierarchical file system name space. SFS, the Secure File System [31], embeds a public key in the name of a file to generate self-certifying pathnames. The idea of extending file system services through namespace is further generalized in Active Names [2].

## 7 Conclusions

We have proposed an active storage system that allows flexible migration of application level code to storage devices. The multiview storage system, MVSS, provides multiple views of an underlying file, similar to multiview database systems providing multiple views of the underlying database. We have shown that it is possible to retain the block-based interfaces of storage devices while allowing flexible service deployment within existing file systems without significant changes to operating systems. We also showed that it is possible to build intelligent devices without porting significant amounts of file system functionality onto disks. Results from a Linux PC-based prototype system demonstrated the effectiveness of our approach. We

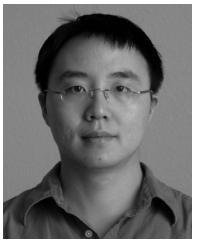
investigated many architectural issues such as the impact of system level parameters including interconnect bandwidth, number of devices, and relative processing power of hosts and disks. Our evaluation has also shown that, while it is possible for individual applications to exploit active storage systems, more work needs to be done for exploiting such systems under mixed workloads.

## ACKNOWLEDGMENTS

This research is funded in part by a US National Science Foundation Career Award, US National Science Foundation research grants CCR-9901640 and CCF-0098263, by the State of Texas, and by EMC Corporation.

## REFERENCES

[1] J. Hartman, U. Manber, L. Peterson, and T. Proebsting, "Liquid Software: A New Paradigm for Networked Systems," Technical Report TR96-11, Dept. of Computer Science, Univ. of Arizona, Tucson, 1996.
[2] A. Vahdat, T. Anderson, and M. Dahlin, "Active Names: Programmable Location and Transport of Wide-Area Resources," *Proc. USENIX Symp. Internet Technologies and Systems,* Oct. 1999.
[3] A. Banchs, W. Effelsberg, C. Tschudin, and V. Turau, "Multicasting Multimedia Streams with Active Networks," Technical Report 97-050, Int'l Computer Science Inst., Berkeley, Calif., 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,* vol. 32, no. 4, pp. 55-56, Apr. 1999.
[5] R.W. Horst, "TNet: A Reliable System Area Network," *IEEE Micro,* vol. 15, no. 1, pp. 37-45, Feb. 1995.
[6] G.A.G. et al., "File Server Scaling with Network-Attached Secure Disks," *Proc. ACM SIGMETRICS Conf.,* pp. 272-284, June 1997.
[7] R.V. Meter, "A Brief Survey of Current Work on Network Attached Peripherals (extended abstract)," *Operating Systems Rev.,* vol. 30, no. 1, pp. 63-70, Jan. 1996.
[8] R.V. Meter, G. Finn, and S. Hotz, "VISA: Netstation's Virtual Internet SCSI Adapter," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 71-80, Oct. 1998.
[9] A. Acharya, M. Uysal, and J. Saltz, "Active Disks: Programming Model, Algorithms and Evaluation," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 81-91, Oct. 1998.
[10] E. Riedel, G. Gibson, and C. Faloustos, "Active Storage for Large-Scale Data Mining and Multimedia," *Proc. 24th Very Large Databases Conf.,* pp. 62-73, Aug. 1998.
[11] K. Keeton, D.A. Patterson, and J.M. Hellerstein, "The Case for Intelligent Disks (IDISKs)," *SIGMOD Record,* vol. 27, no. 3, pp. 42-51, Sept. 1998.
[12] D.S.H. Rosenthal, "Requirement for a "Stacking" vnode/vfs Interface," Unix Int'l document SD-01-02-N014, 1992.
[13] J.S. Heidemann and G.J. Popek, "File System Development with Stackable Layers," *ACM Trans. Computing Systems,* vol. 12, no. 1, pp. 58-89, Feb. 1994.
[14] J.B.C. et al., "Impulse: Building a Smart Memory Controller," *Proc. Fifth Int'l Symp. High Performance Computer Architecture,* pp. 70-79, Jan. 1999.
[15] M. Sivathanu, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Evolving rpc for Active Storage," *Proc. 10th Symp. Architectural Support for Programming Languages and Operating Systems,,* pp. 264-276, Oct. 2002.
[16] S. Cannon, "Concurrent File System-Making Highly Parallel Mass Storage Transparent," *Proc. Int'l Conf. Supercomputing,* Apr. 1989.
[17] J. Heidemann and G. Popek, "Performance of Cache Coherence in Stackable Filing," *Proc. 15th Symp. Operating Systems Principles,* pp. 127-142, Dec. 1995.
[18] W. Almesberger, A. Kuznetsov, and J.H. Salim, "Differentiated Services on Linux," *Proc. GlobeCom,* pp. 831-836, Dec. 1999.
[19] M. Blaze, "A Cryptographic File System for Unix," *Proc. First ACM Conf. Comm. and Computing Security,* pp. 9-16, Nov. 1993.
[20] E. Zadok, "Cryptfs: A Stackable vnode Level Encryption File System," Technical Report CUCS-021-98, Computer Science Dept., Columbia Univ., 1998.
[21] B. Schneier, *Applied Cryptography.* second ed. New York: John Wiley and Sons, pp. 336-339, 1996.
[22] G. Memik, M.T. K, and A.N. Choudhary, "Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads," *Proc. Int'l Conf. Parallel Processing,* pp. 335-342, Aug. 2000.
[23] M. Uysal, A. Acharya, and J. Saltz, "Evaluation of Active Disks for Decision Support Databases," *Proc. Eighth Int'l Symp. HPCA,* pp. 337-348, Jan. 2000.
[24] R.V. Meter, S. Hotz, and G. Finn, "Derived Virtual Devices: A Secure Distributed File System Mechanism," *Proc. Fifth NASA Conf. Mass Storage Systems and Technologies,* Sept. 1996.
[25] C.R. Atanasio, M. Butrico, C.A. Polyzois, S.E. Smith, and J.L. Peterson, "Design and Implementation of a Recoverable Virtual Shared Disk," Technical Report RC 19843, IBM, Nov. 1994.
[26] W. de Jonge, M.F. Kasshoek, and W.C. Hsieh, "The Logical Disk: A New Approach to Improving File Systems," *Proc. 14th ACM Symp. Operating Systems Principles,* pp. 15-28, Dec. 1993.
[27] E.K. Lee and C.A. Thekkath, "Petal: Distributed Virtual Disks," *Proc. Seventh Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 84-92, Oct. 1996.
[28] G.C. Skinner and T.K. Wong, "'Stacking' vnodes: A Progress Report," *Proc. Summer USENIX Technical Conf.,* pp. 161-174, June 1993.
[29] E. Zadok, "Fist: A Language for Stackable File Systems," *Proc. USENIX Technical Conf.,* June 2000.
[30] D.K. Gifford, P. Jouvelot, M.A. Sheldon, and J. O'Toole, "Semantic File Systems," *Proc. ACM Symp. Operating Systems Principles,* pp. 16-25, Oct. 1991.
[31] D. Mazieres, M. Kaminsky, M.F. Kaashoek, and E. Witchel, "Separating Key Management from File System Security," *Proc. ACM Symp. Operating Systems Principles,* pp. 124-139, Dec. 1999.



**Xiaonan Ma** received the BS degree in electrical engineering from Nanjing University, Nanjing, China, in 1996, and the PhD degree in electrical engineering from Texas A&M University, College Station, TX, in 2002. He joined the IBM Almaden Research Center, San Jose, Calif., as a Research Staff Member in 2002. His current research interests are in the areas of advanced storage systems and distributed file systems. He is a member of the IEEE Computer Society.

**A.L. Narasimha Reddy** received the BTech degree in electronics and electrical engineering from the Indian Institute of Technology, Kharagpur, India, in August 1985, and the MS and PhD degrees in computer engineering from the University of Illinois at Urbana-Champaign in May 1987 and August 1990, respectively. At the University of Illinois at Urbana-Champaign, he was supported by an IBM Fellowship. He is currently an associate professor in the Department of Electrical Engineering at Texas A&M University. He was a research staff member at IBM Almaden Research Center in San Jose from August 1990-August 1995. Reddy's research interests are in storage systems, multimedia, network, and computer architecture. Currently, he is leading projects on building network-based storage systems and partial-state-based network elements. While at IBM, he coarchitected and designed a topology-independent routing chip operating at 100 MB/sec, designed a hierarchical storage management system, and participated in the design of video servers and disk arrays. He is a member of the ACM SIGARCH and is a senior member of IEEE Computer Society. He received a US National Science Foundation CAREER Award in 1996. He recieved an outstanding professor award at Texas A&M during 1997-1998.