Implementation and Evaluation of an Active Storage System Prototype

Xiaonan Ma, A. L. Narasimha Reddy

Abstract— **This paper presents the implementation and evaluation of an active storage system. Our prototype system is based on the multi-view storage system (MVSS). It uses the notion of views to support "filtering" of data at the storage devices. Similar to views of a database in a multi-view database system, views in our system are generated dynamically and are not stored on physical storage devices. Each view of an underlying file is represented through a separate entry in the file system namespace. Our system is based on the existing device level interface and requires minimal change to file systems and OS. This paper presents the key features and an evaluation of the prototype system built on Linux-based PCs. We provide results from example applications implemented on the prototype and discuss the various issues in optimizing the performance of such a system in mixed workloads.**

Keywords— **Active storage, virtual block address, mixed workloads**

I. Introduction

ACTIVE storage has been proposed to migrate application-specific processing to storage devices [1], [2], [3]. With the proliferation of Server Area Networks (SANs) and the growing use of network attached disks (NADs), employing active storage systems with NADs provides an attractive architecture that scales with the increasing size of datasets. Offloading computation closer to the data source offers benefits such as more computing power at little extra cost, potential reduction in data movement through the I/O subsystem on host as well as the interconnect network, and better scalability. However, existing approaches either requires porting file systems onto the devices (which is difficult for NADs based a block interface, such as iSCSI) or extensive modification to the OS (particularly the file system).

Previous work [1], [2], [3], [4] has provided significant evidence that single applications such as database decision support can benefit from active storage systems. In real-world systems, the storage devices may need to continue supporting other normal I/O operations while allowing multiple applications to exploit the active storage simultaneously. The effect of a combination of these workloads has not been previously studied.

In this paper, we will present the implementation and evaluation of an active storage system that can be easily supported with commercial file systems. Our system is based on the multi-view storage system (MVSS) [5]. MVSS provides a scheme to separate the deployment of services from file system implementations. Similar to a multi-view database system [6], MVSS allows multiple views (also called virtual files in this paper) of the same file on the physical disk to be generated dynamically. Multiple views of the file are provided to the user through file system namespace. Through these views, MVSS provides a flexible and extensible way for supporting various device-level enhancements. In this paper, we concentrate our focus on active storage where data is processed or filtered close to the devices before returned to the host. We report on various system level issues that impact performance in such a system and provide an evaluation of active storage systems in mixed workloads.

II. Active storage architecture framework

MVSS was designed to keep the software layer on the disk as thin as possible and to minimize changes to existing operating systems. To keep the paper self-contained, we provide a brief overview of MVSS as it pertains to our active storage system prototype 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 underlying file (called base file) and certain application level processing service to be carried out at the storage device. For example, a base MPEG file may have multiple views corresponding to different levels of quality. Each virtual file has its own pathname, in-core inode and uses separate buffers in the system. To support this, MVSS

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

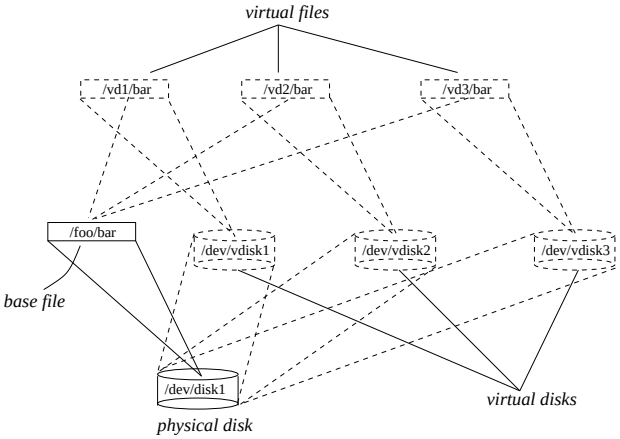


Fig. 1. Concept of virtual files in MVSS

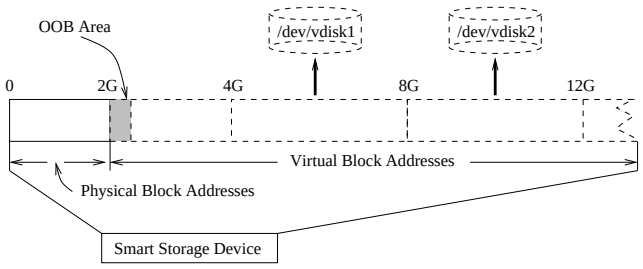


Fig. 2. An example of virtual block address space management in MVSS

introduces the concept of virtual disks. Virtual disks are place holders for virtual files. 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 I/O requests sent to the virtual disk to be forwarded to the underlying device. Mounting a virtual disk creates virtual namespace for files on the device that the virtual disk is hooked to. 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 are hooked to disk `/dev/disk1`. Mounting these virtual disks then 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. We developed the multi-view file system (MVFS) for management of virtual files. MVFS is a stackable file system layer based on the vnode structure [7]. It sits on top of the native file systems and forwards them file system operations such as name resolution. Using a stackable file system layer allows us to add new features on top of existing file system codes conveniently with little overhead.

Second, a flexible user interface — *attach* is provided to allow users to dynamically associate services with each view of a file. *Attach* has the following interface: *attach(virtual file, service name, parameters). Service name* specifies the service, such as encryption, compression, etc. Examples of *parameters* include keys for encryption, query criteria for database SELECT operation, quality of service (QoS) parameters for MPEG filters, etc. 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 the need for modification of existing application codes. 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.

Third, we employ a smart storage device model based on the common block-level interface. Application-specific service information is bound with I/O requests through the *virtual block addresses* of devices. Virtual block addresses are block addresses beyond the physical capacity of the device. The idea of using virtual block addresses is based on the following observation: capacities of block devices are usually much less than the maximum value that the operating system could support. The use of virtual block addresses is compatible with existing buffer cache schemes and allows simple reuse of existing file system and operating system technology. 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 and can be dynamically adjusted. 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. Figure 2 shows an example of a device's virtual block address space. Note that the *attach* interface allows users to associate services with virtual files at the file level, but the service binding information still needs to be passed down to the device level, where the services are performed. However, the concept of files is not available at the device level. Introducing new interfaces between the file system and the device requires substantial modification to existing file systems and buffer cache schemes. To solve

this problem, each virtual file is allocated some virtual blocks when it is accessed for the first time. These virtual blocks are associated 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. Our system 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. Each device reserves a certain range of virtual block addresses for this, called the OOB area, which is also used for uploading application-specific filters onto the device. The replicated virtual block maps and AUX areas enable 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.

III. Architectural Issues

There are two main advantages of active disks: (1) increased parallelism through the employment of multiple processors at the device, (2) reduced data movement on the I/O interconnect network through filtering data close to the devices. The performance of an application in an active storage system is affected by: (1) host and disk processing capacity, (2) device I/O capacity and (3) interconnect bandwidth. In order to understand the impact of the various system parameters on performance, we explore a number of experimental settings by varying the CPU speed, interconnect bandwidth and the number of disks in the system.

In traditional storage systems, devices provide no distinction between data requests from different applications. In an active storage system, different active data requests require different processing on the device. Previous studies have clearly shown the advantage of the active storage systems in single application scenarios. However, assigning all active requests with different services the same priority may not achieve the best overall system performance. This brings out the following question: how should normal data requests and active data requests be scheduled at the device? To evaluate the relative advantages of active storage systems compared to normal systems in mixed workload environments, we explore the performance issues in a mixed workload environment with both active and normal data requests. In our system, active requests can be distinguished by the block addresses (active requests use virtual block addresses beyond the normal address range). We expect that this evaluation will lead to a better understanding of active storage systems in general.

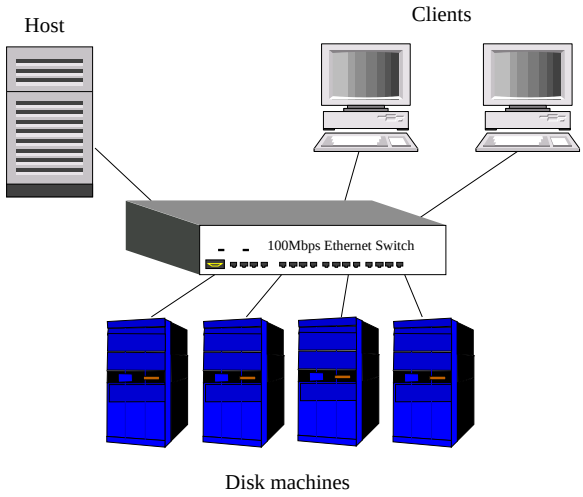


Fig. 3. Testbed structure

IV. Implementation

The structure of our testbed is shown in figure 3. 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. We use the Linux Network Block Device (NBD) driver on the host to communicate with the smart disks. NBD allows 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

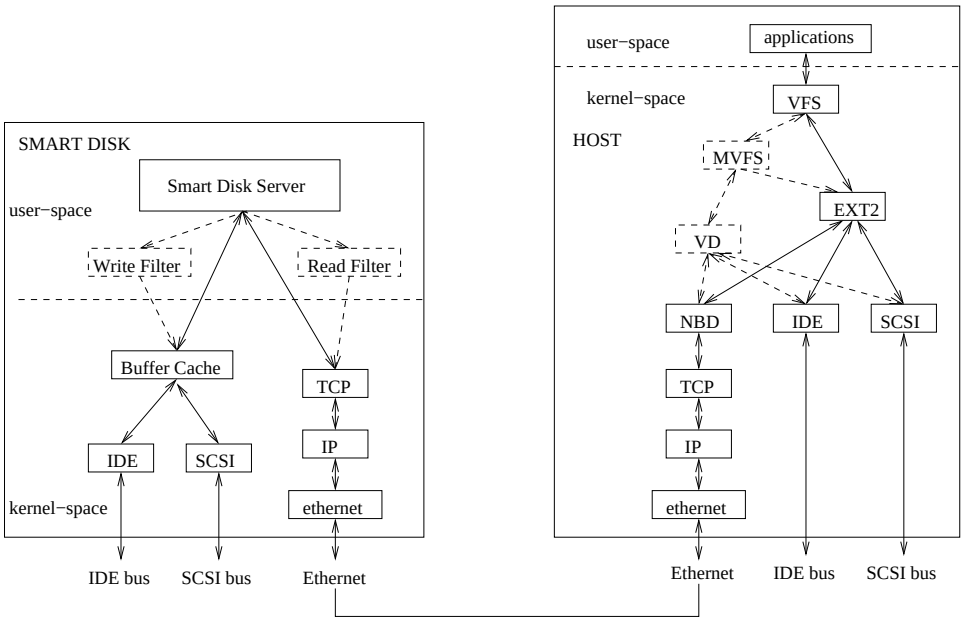


Fig. 4. MVSS structure

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. We also used the Linux kernel network QoS support [8] to simulate different link speeds between the host and the devices. 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.

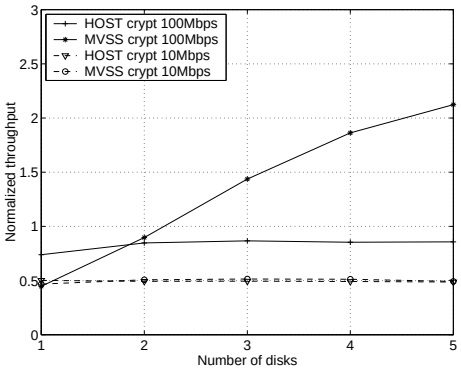
Figure 4 shows the structure of our system. Solid lines illustrate the structure and data path of a traditional system with NBD. Dashed lines show the new components and data path in MVSS. On the host side, we modified the NBD device driver slightly to support virtual block requests. We implemented both MVFS and the virtual disk driver (VD) as loadable kernel modules. MVFS manages the AUX areas for virtual files and the virtual block maps for virtual disks. File system operations such as name resolution are forwarded to native file system (e.g., EXT2). The virtual disk driver forwards I/O requests to the underlying device driver. Data requests are 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.

Because of 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. 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.

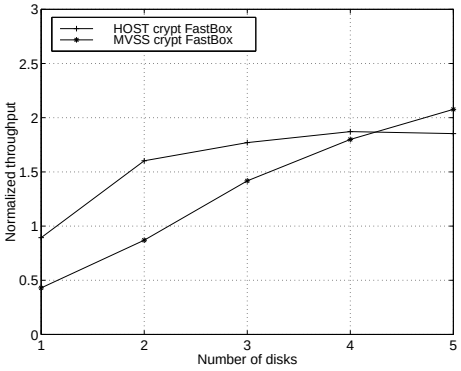
V. Applications and Results

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

Blowfish Crypt Cryptographic techniques are becoming increasingly important in modern computing



(a) Results with 233MHz host



(b) Results with 500MHz host

Fig. 5. Performance with the Blowfish encryption applet

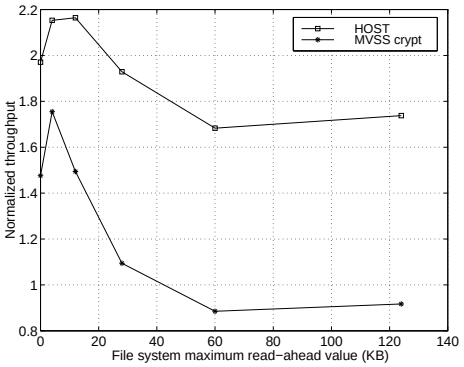
system security. However, user-level tools are usually cumbersome. Adding cryptographic support at system level provides better transparency [9]. With the current trends in storage pooling and outsourcing, data are increasingly being stored encrypted on the device. Secure storage at system level can 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 [10]. 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 5(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 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).

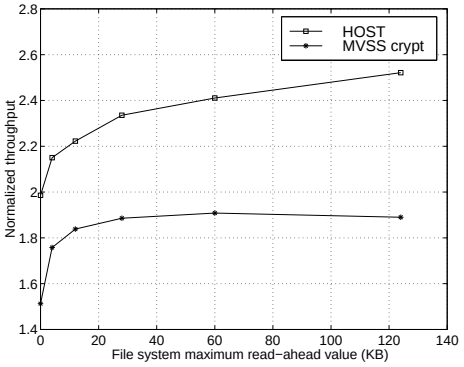
Figure 5(a) also shows the results when the interconnect bandwidth was set to 10Mbps. The results for HOST crypt and MVSS crypt are almost identical. Both are limited by the low interconnect bandwidth. The results show that the ability to exploit the CPU power on the disks could be severely limited when 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.

Figure 5(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 primar-

ily dependent on processing power and does not see any gains through reductions in data traffic across the I/O interconnect fabric.



(a) Request queue of length 128



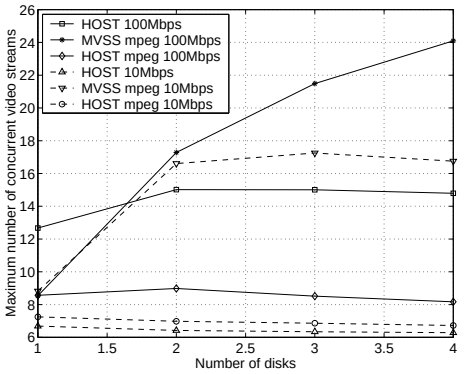
(b) Request queue of length 2048

Fig. 6. Performance with different file system read-ahead values

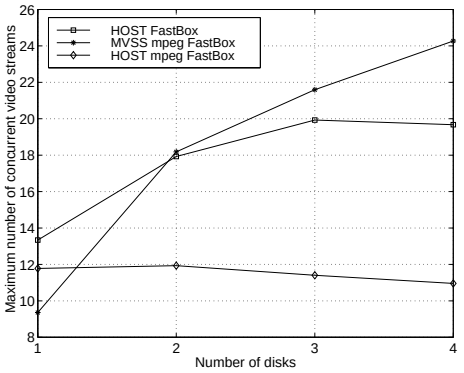
During our experiments with multiple disks, we noticed that large file system read-ahead (or prefetch) values resulted in less total throughput for the system than small read-ahead values. Further investigation showed that the reason for this is that Linux uses a common fixed length request queue for 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 6(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 5(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 6(b) shows the result after we increased the length of the request queue to 2048 requests.

MPEG QoS Filtering 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.

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. 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 in it. Figure 7(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



(a) Results with 233MHz host



(b) Results with 500MHz host

Fig. 7. Performance with MPEG application

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.

Figure 7(b) gives the results when we replaced the 233MHz host with an faster machine that has a 500MHz CPU and 128MB memory. The active storage system still provides higher throughput with two or more disks. 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.

Median Filtering In previous experiment, 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 median filter 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 8 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.

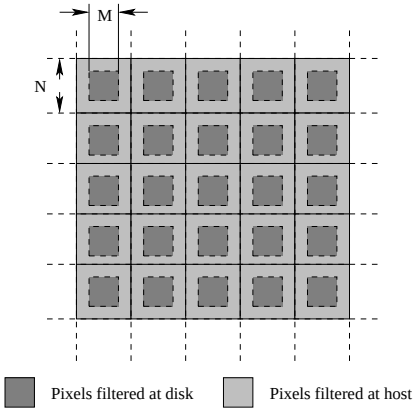


Fig. 8. Median filter partition

In this experiment, we developed a simple median filter using a 3 by 3 kernel. We formatted the input image file so that the sub-image size is equal to the block size for convenience. Figure 9 shows the results

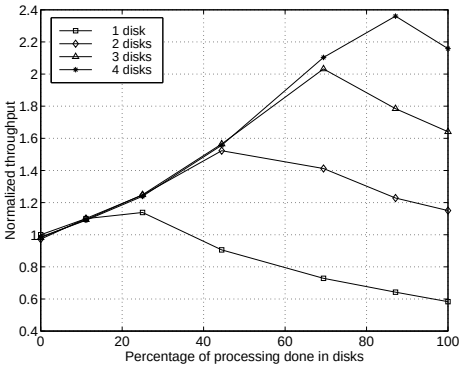


Fig. 9. Performance with the median filter

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.

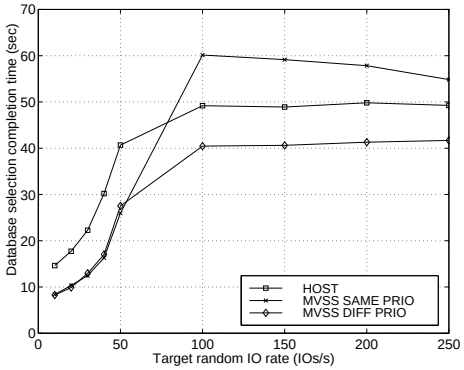
Since most file systems automatically issue read-ahead requests, the advantage of parallelism between disks and host can be exploited by a normal single thread application (as the user application we ran on the host in this test).

VI. Mixed Workload

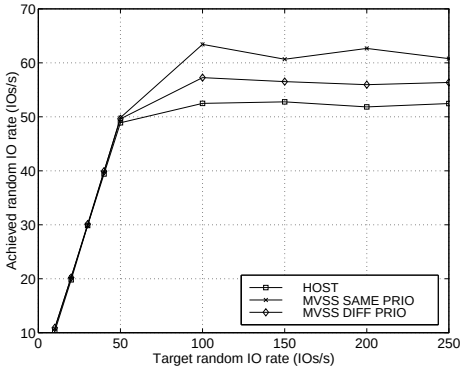
To evaluate the performance of our system in mixed workloads, we created a workload of random I/Os along with active data requests from a database SELECT application. We generated datasets with 64-bytes long records and developed a simple filter applet to filter records based on a selectivity parameter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

Random I/O requests are generated to retrieve data blocks at random locations on the disk. These I/O requests are normal read requests of 4KB each. In the experiment, we let ten 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 5 to 250 I/Os per second. The achieved random I/O rate may be lower than the target rate when the request rate is beyond what the disk can handle.

For these workloads, we considered the case when two separate threads were used for active and normal requests respectively. In the first setting, we gave both



(a) Database select completion time

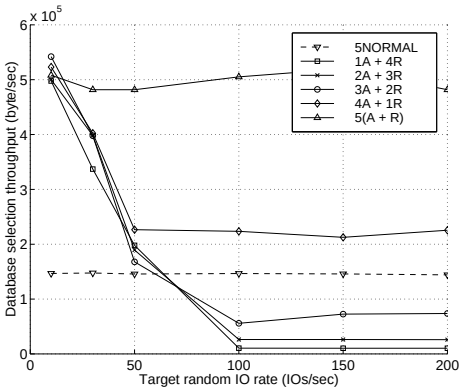


(b) Achieved random I/O rate

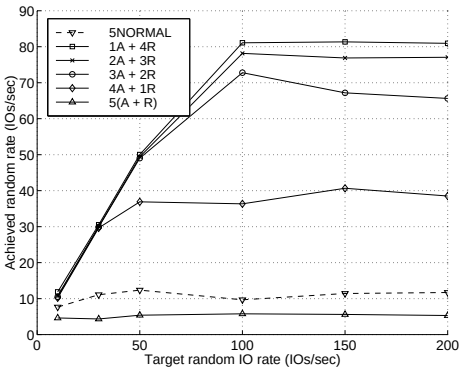
Fig. 10. Mixed workload with two threads given different priorities (100Mbps interconnect BW)

threads identical priorities. In the second setting, we gave the active thread higher priority over the normal thread. We compare the results against that of 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 figure 10(a) give the time taken by the database select application to finish processing a 32MB dataset with different target random I/O rate (the database application simple issues requests as fast as possible). Figure 10(b) shows the actual number of random I/Os completed during the same period. We did the test with selectivity of 8 for the database applet and 100Mbps interconnect bandwidth. When the active request thread is given a higher priority over the normal request thread (MVSS DIFF PRIO), the database select 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. In MVSS DIFF



(a) Database select throughput



(b) Achieved random I/O rate

Fig. 11. Mixed workload with totally five threads (10Mbps interconnect BW)

PRIO, the database application finishes faster and at the same time achieves a higher random I/O rate than the normal storage system (HOST). It is observed that the database application finishes faster in the normal system (HOST) than MVSS with equal priority (MVSS SAME PRIO) when the target random I/O rate is beyond 100 IOs/sec. This is because a higher random I/O rate is achieved in the latter case. There the random I/O requests are processed separately with active requests, which means that a random request may be completed before the active requests that arrive the disk earlier than it.

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 normal request, upon the completion of the thread's earlier request. In other combinations, threads were dedicated to either handle active requests or 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 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 five threads through a common FIFO queue. We limited the interconnect bandwidth to 10Mbps in the experiment. Figure 11(a) shows the database select throughput while figure 11(b) shows the achieved random I/O rate. 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 handle 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 rate. With four active request threads and one random request thread (4A+1R), the active storage system achieves higher database select throughput and random I/O rate at all random I/O rates. In other cases where the number of active request threads are close to or less than that of random request threads, the database select throughput of the active storage system becomes lower than that of the normal system (5 NORMAL) when 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 request threads, since the random request rate is much lower than the active request rate in the experiments. Compared with those in figure 10, the results here also show that the decrease in interconnect bandwidth from 100Mbps to 10Mbps 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. The relative priorities between the active and normal I/O threads affect the their relative performance. If some normal I/O requests are of interactive nature, it seems best to dedicate at least one thread for serving them at the disk in mixed workloads. It is possible to employ more sophisticated scheduling algorithms to control the performance of

different applications. This is a subject for future work.

VII. Related Work

The Active Disk and IDisk models [1], [2], [3] have been proposed 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 [4] for active disks against two alternative architectures: shared memory multiprocessors (SMPs) and workstation clusters. Our work draws much inspiration from these earlier papers but focuses on a real implementation and how to exploit the benefits of active storage devices within the existing storage systems. Our system supports a block-level interface unlike the stream model proposed in earlier approaches. The scheduling and mixed workload issues are not addressed in these earlier work.

Stackable file system allows extension of functionalities for existing file systems through Vnode Stacking [7], [11]. Earlier work on stackable file system has been focused on file level enhancement and does not support service migration to the devices.

VIII. Conclusions

We have implemented an active storage system that allows flexible migration of application level code to the storage devices. 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. Our evaluation has shown that while it is possible for individual applications to exploit active storage systems, much work needs to be done for exploiting such systems in mixed workloads.

References

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

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

[3] 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.

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

[5] X. Ma and A. L. Narasimha Reddy, "MVSS: Multiview Storage System," *Proc. of ICDCS*, Apr. 2001.

[6] J. O. Ullman, *Principles of database systems*, Computer Science Press, 1984.

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

[8] Werner Almesberger, Alexey Kuznetsov, and Jamal Hadi Salim, "Differentiated services on linux," in *Proceedings of Globecom'99*, December 1999, pp. 831–836.

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

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

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