Evaluation of Active Storage Strategies for the Lustre Parallel File System

Juan Piernas
Pacific Northwest National Laboratory
P.O. Box 999
Richland, WA 99352
juan.piernascanovas@pnl.gov

Jarek Nieplocha
Pacific Northwest National Laboratory
P.O. Box 999
Richland, WA 99352
jarek.nieplocha@pnl.gov

Evan J. Felix
Pacific Northwest National Laboratory
P.O. Box 999
Richland, WA 99352
evan.felix@pnl.gov

ABSTRACT

Active Storage provides an opportunity for reducing the amount of data movement between storage and compute nodes of a parallel filesystem such as Lustre, and PVFS. It allows certain types of data processing operations to be performed directly on the storage nodes of modern parallel filesystems, near the data they manage. This is possible by exploiting the underutilized processor and memory resources of storage nodes that are implemented using general purpose servers and operating systems. In this paper, we present a novel user-space implementation of Active Storage for Lustre, and compare it to the traditional kernel-based implementation. Based on microbenchmark and application level evaluation, we show that both approaches can reduce the network traffic, and take advantage of the extra computing capacity offered by the storage nodes at the same time. However, our user-space approach has proved to be faster, more flexible, portable, and readily deployable than the kernel-space version.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Systems and Software; H.3 [**Information Storage and Retrieval**]: Information Storage

General Terms

Design, Performance

1. INTRODUCTION

Recent improvements in storage technologies in terms of capacity as well as cost effectiveness, and the emergence of high-performance interconnects, have made it possible to build systems of unprecedented power by connecting thousands of compute nodes and storage nodes. However, for large-scale scientific simulations that use these environments, the efficient management of enormous and increasing volumes of data remains a challenging problem. Despite the improvements of storage capacities, the cost of bandwidth for moving data between the processing nodes and the storage devices has not improved at the same rate as the disk capacity. One approach to reduce the bandwidth requirements between storage and compute devices is, when possible, to move computation closer to the storage devices. Similarly to the processing-in-memory (PIM) approach for random access memory [16], the *active disk* concept was proposed for hard disk storage systems [1, 15, 24]. The active disk idea exploits the processing power of the embedded hard drive controller to process the data on the disk without the need for moving it to the host computer. Unfortunately, the active disk ideas have not been fully adopted by the hard drive vendors or led to practical implementations.

In recent years, several proprietary parallel/distributed file systems (Google's GFS [10], IBM's GPFS [13], Panasas ActiveScale File System [21], SGI's CXFS [27]), and open source ones (Lustre [5], PVFS [4], RedHat's GFS [22]), have been developed to tackle the problem of data management in the context of high-performance computing systems, and other high data volume environments. Some of these file systems, such as Google's GFS, Lustre and PVFS, use mainstream server computers as storage nodes, that is, computers that contain significant CPU and memory resources, have several disks attached to them, and run a general-purpose operating system (usually Linux). In many cases, the server nodes of the parallel file system are very similar or identical to the compute nodes deployed in a cluster, and offer Giga-op/s of processing power. The combined computing capacity of these hundreds, or even thousands, of storage nodes can be considerable. However, it is not usually exploited due to the role of these nodes as I/O elements that only store data.

One approach to take advantage of the underutilized CPU time in the storage nodes is to extend the traditional active disk concept to the parallel file systems of modern high-performance architectures. We call this approach **Active Storage** in the context of the parallel file systems. Two important differences with respect to the previous work are that (1) storage devices are now full-fledged computers, and (2) they include a feature-rich environment provided typically by a Linux operating system. These two factors make it possible to run regular application codes on the storage nodes. On the other hand, by offloading some computing tasks to the storage nodes, near to the data that they manage, Active Storage makes it possible to substantially reduce

SC '07 November 10-16, 2007, Reno, Nevada, USA

the data movement across the network and, hence, the overall network traffic. More specifically, by performing data processing at the source, data does not need to be moved between filesystem servers and cluster compute nodes.

Active Storage is targeted at applications with I/O-intensive stages that involve fundamentally-independent data sets. For example, it can be used to process, either on-line or off-line, output files from scientific simulation runs. Some examples of tasks suitable for Active Storage include: compression and archival of output files, statistical analysis of the output data and storing the results in an external database, indexing the contents of the output files, simple data transformations such as unit conversion by multiplying a set of numbers by a scalar, etc. By performing these operations in the storage nodes, we not only achieve the aforementioned benefits with respect to the resource usage, but also can exonerate scientific application programmers from implementing I/O tasks which are "oblivious" to the main application. For the Active Storage concept to become widely adopted, we need to develop low-overhead, scalable, and flexible implementation mechanisms that allow applications to express a variety of algorithms and related data processing patterns.

In this paper we present and evaluate two implementations of the Active Storage concept in the context of the Lustre 1.6 file system. Lustre is a popular open source parallel filesystem that, among others, has been selected by Cray for their products, and will be available on their upcoming petascale systems to be installed in the ORNL National Leadership Computing Facility. The first implementation of Active Storage considered in the present paper is based on a previous implementation by Felix *et al.* [9] for an older 1.4 version of Lustre. This implementation relies on a kernel module which interacts with other kernel modules of Lustre. The second implementation considered in the paper is our new contribution to the field, and it proposes a pure user-space implementation of the Active Storage concept. In both approaches, the application code runs in user-space, like any other regular process.

We have benchmarked two applications in three different configurations: without Active Storage, with the kernel-space implementation of Active Storage, and with the user-space one. We have also analyzed the overall execution time, the completion time from a client point of view, and the generated network traffic. The experimental results show that both Active Storage approaches can reduce the network traffic to near zero for some workloads, and can take advantage of the extra computing capacity offered by the storage nodes at the same time. However, our proposed user-space approach has proved to be more flexible, portable, and readily deployable than the kernel-space counterpart. For example, we have also ported it to the PVFS file system with minor changes, and the results obtained have been similar to those presented here. In addition, the proposed user-space approach is competitive with the traditional approach. For example, for a bioinformatics application example running on a Linux cluster with Myrinet and using four storage servers, the execution time was reduced by 51.8% over the kernel space approach, and 69.3% over the version of the code that did not use Active Storage.

The rest of the paper is organized as follows. Section 2 provides an overview of work related to the *active disks* concept. Section 3 describes the kernel-space approach of Active Storage and its implementation in Lustre. Section 4 details our user-space proposal and lists its main advantages over the previous one. Experimental results regarding execution time and network traffic are presented in Section 5. Conclusions and some future directions appear in Section 6.

2. RELATED WORK

The idea of *intelligent storage* was developed by several authors at the end of the 90's, with some similar ideas proposed even earlier in the 80's in the database world [6]. The original research efforts were based on the premise that modern storage architectures have progressed to a point that there exist the real possibility of utilizing the unused processing power provided by the drive controller itself. However, for numerous reasons, commodity disk vendors have not offered the required software support and interfaces to make intelligent storage practical and widely used.

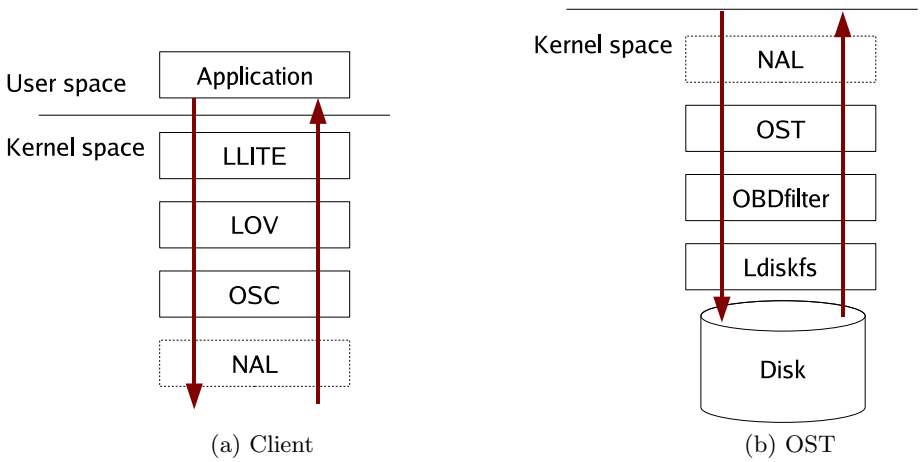
Riedel *et al.* [24] propose a system, that they call *Active Disks*, which takes advantage of the processing power on individual disk drives to run application-level code. Since they propose to use the processors embedded in individual storage devices to run both application and "core" drive codes, they need to define a framework inside the drive to ensure proper execution of code and safeguard the integrity of the drive. Our work on Active Storage, however, focuses on parallel filesystems with commodity disk drives, general purpose CPU, and operating systems.

Keeton *et al.* [15] present a computer architecture for decision support database servers that utilizes "intelligent" disks (IDISKs). IDISKs use low-cost embedded general-purpose processing, main memory, and high-speed serial communication links on each disk. Their proposal is similar to that presented in the previous paragraph, although they also include a high-speed interconnect to overcome the I/O bus bottleneck of conventional systems and provide a scalable I/O subsystem.

Acharya *et al.* [1] evaluate *active disks* architectures, and propose a stream-based programming model which allows *disklets* (disk-resident code of applications) to be executed efficiently and safely on the processors embedded in the disk drives. Disklets take streams as inputs and generate streams as output. Files (and ranges of files) are represented as streams. In our Active Storage work, the processing components also read and write data by using streams, although, unlike disklets, they are full-fledged user-space processes which, therefore, are only limited by the policies enforced by the operating system.

By using the concept of active disks, and representing files as objects, Lim *et al.* [17] propose an Active Disk File System (ADFS) where many of the responsibilities of a traditional central file server, such as authentication, pathname lookup, and storage space management, are offloaded to the active disks. These authors suggest that objects (files) can have application-specific operations which can be run by disk processors, so that only the results are returned to clients. However, they do not provide or implement the proper framework to do that.

Another concept which profits the processing power of the disk drives, and delegates more responsibilities to them, is the *object-based storage device* (OSD) [11, 18]. The OSD's have become a key element in some recent parallel file systems (such as Lustre [5] and the Panasas File System [21]), other industry products (IBM Storage Tank [12], EMC Centera [8], etc.), and new-design storage systems [20, 28]. The



(a) Client (b) OST

Figure 1: Lustre layers in a client and storage node

interface of the object-based storage devices has been standardized recently [14, 23]. Some storage device manufacturers, such as Seagate [26], are also developing object-based storage devices.

By using the OSD concept, Schlosser and Iren [25] propose to use and enhance the OSD interface in order to enable better communication between database and storage systems. The idea is that the database application should communicate semantic information (e.g., the geometry of a relation) and other quality of service requirements to the storage subsystem, allowing it to make data allocation decisions on behalf of the application. They also suggest that, with more processing capability in the storage devices, it would be possible to delegate some database-specific tasks (e.g., indexing, search and retrieval) to the OSDs, in an active disk fashion.

Du [7] merges the OSD and active disk concepts in order to build the *intelligent storage devices* one. An intelligent storage device is directly attached to the network (it contains an IP stack and can be easily plugged into an IP network), supports the OSD concept, supports the active disk concept (it has an embedded processor and some amount of memory which allow it to carry out some limited tasks), and implements a Global Unique ID (GUID) for each file (therefore, it makes the management of distributed replicas much easier).

Finally, Braam and Zahir [3] also mention a possible connection between the idea of active disks and the Lustre file system, but have not implemented it.

Some of the main shortcomings of the previous studies are: they lack a real implementation, the processing capacity and memory are limited to that available in the storage device, the solution is specific to a problem or application, or the code to run in the active disk is confined to a restricted domain which is much less powerful than the one offered by a general-purpose operating system.

Felix *et al.* [9], however, present a first real implementation of *Active Storage* for the Lustre file system. That implementation provides a kernel-space solution with the processing component parts implemented in the user space. The kernel-space part is a module that integrates with other Lustre modules in the storage nodes. The module makes a copy of the data which arrives from the clients, and passes it to the user-space component. This component then processes the data, and write it back to the storage device via the module. Since the code is run in the user-space of a general-purpose operating system, on a computer with significant CPU and memory resources, this solution does not suffer the aforementioned problems.

Our proposal is based on the *Active Storage* concept, but it offers a solution that is purely in user space. As the the following sections of the paper show, this makes our approach more flexible, portable, and readily deployable than the existing one, while it achieves the same, or even better, performance. The portability advantages of our approach are demonstrated not just for Lustre but also through a port to the PVFS filesystem.

3. ACTIVE STORAGE IN KERNEL SPACE

Lustre [5] is implemented as a stack of Linux kernel modules which support the different objects that appear in a typical Lustre installation. In this layered architecture, a layer can implement almost anything: a simple pass-through inspection layer, a security layer, or a layer which makes large changes to the behavior of the file system. Our first approach for Active Storage exploits this modular structure of Lustre to implement a new object (e.g., a new module) in order to provide the desired functionality. This approach is based on the Active Storage implementation by Felix *et al.* [9], although there are some differences that will be described later.

The classic Lustre client layers are shown in Figure 1-a. The client sees a file system that supports POSIX semantics for reading and writing files. Once the kernel system calls are made, the kernel passes control to the *llite* layer. This mostly translates the basic Virtual File System (VFS) calls into work orders for the rest of the system. It has direct communication with the Meta Data Client (MDC), the ManaGement Client (MGC), and the Logical Object Volume (LOV) modules. The LOV layer is responsible for dispatching the data related calls to the set of Object Storage Clients (OSC) that are available. It performs these duties based on striping information obtained from the Meta Data Server (MDS). The OSC layer packages up the requests and sends them to the Object Storage Servers (OSS), serving the Object Storage Targets (OST) that are specified, over the Portals Network Abstraction Layer (NAL) [2].

The OSS's serve a set of OST's. These targets relate di-

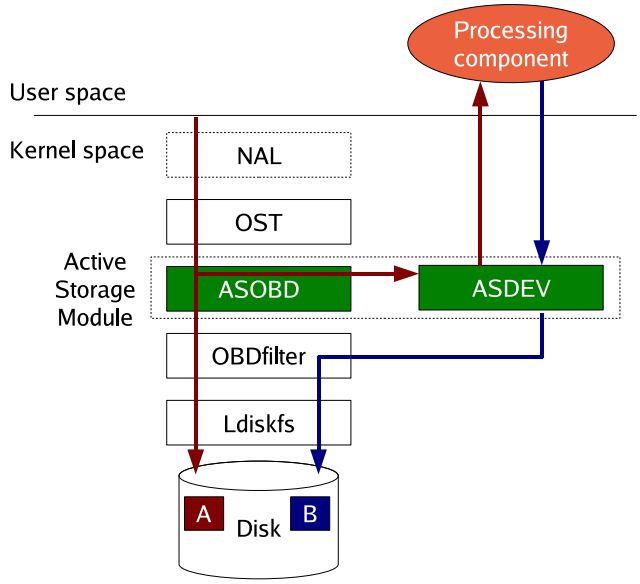


Figure 2: Kernel-space implementation of Active Storage in Lustre

rectly to the underlying available disk file systems, as we can see in Figure 1-b. The OST layer receives the requests from the NAL, and feeds them down to the OBD filter layer. This OBD filter wraps the Lustre disk file system, *ldiskfs* (a particular version of the Linux Ext3 file system), so that it looks like an Object Based Disk.

The Active Storage (AS) layer is attached between the OST layer and the OBD filter layer. Once in place, the module acts like a pass-through module, with roughly the same behavior as the OBD filter, until such time an Active Storage task is requested. To initiate the Active Storage process, a client application will create empty files, and then send a special command to the file system specifying the Object ID's of all input and output files, along with information relating to the type of processing needed, and parameters for this processing.

Figure 2 shows the processing that takes place as a file is written, according to the processing pattern supported by the current implementation. For example, the client would create two empty files A and B, and then send an AS linking command. Once this takes place, a device is allocated to an AS process. The Active Storage device module interfaces with a processing component (PC) on the OSS. The processing component is implemented as a standard Linux process that runs on behalf of the user. This process is spawned at the time when the link is made. A small helper process is started that sets up the correct interface, and then starts the PC. After the processing component completes its work and exits, the helper process cleans up the interface files and then terminates.

Once the files have been created, linked, and the processing component initialized, the remote client will start writing data to file A, using normal file API calls. The Active Storage layer sees this data, and passes it directly to the OBD filter layer. It also makes a copy, which is sent to the Active Storage device layer. When the processing component calls a `read()` operation on the device interface, it will either get the waiting data, or will block until new data become available. Once the processing component has completed its work, any output is written out by calling `write()` on the AS device. The AS device layer receives these writes, and feeds them to the OBD filter layer. These writes will all appear in file B. Therefore, the original (unprocessed) data appears in file A, and the results of the AS processing component appear in file B.

The processing component accesses the input and output files as streams, much in the same way as in the Felix's implementation, which in turn resembles the approach described in [1]. One difference between the Felix's implementation and ours is that our processing components determine the transfer size for the input stream, rather than being restricted to receive only one block at a time. By using bigger sizes, it is possible to reduce the number of context switches and improve the performance. Another difference is that our implementation guarantees the byte order in the input stream. In the Felix's proposal, the processing component must take care of the byte order, what makes its implementation more difficult.

Since the processing component has full access to the normal Linux environment, it has access to most of the operating system services, such as network, and local file systems. Therefore, its communication is not confined to the kernel module. For example, it could store the output results in an external database server.

4. ACTIVE STORAGE IN USER SPACE

The underlying idea of Active Storage is that the processing power of the storage nodes can be used to perform computing tasks on data which is read from and written to local files. This can be completely done in user space if the storage nodes are also clients of the parallel file system. In that way, the storage nodes can access to all the files in the file system and, specifically, to the files stored locally.

The initial implementation of Active Storage was done in kernel space for the Lustre 1.4 series, which did not support using a storage node as a Lustre client at the same time. This constraint, however, has been removed from the Lustre 1.6 version.

4.1 Design and Implementation

Figure 3 depicts our new approach. In order to build an Active Storage system, there must be one background process per OST, which plays the same role as the processing component of the kernel-space approach. Each background process reads from a file, does some processing, and then writes out the result to another file. Both the input and output files are stored in the corresponding local OST server. As we can see, the input files contain the output data produced by the remote clients (in our case, the remote clients run in compute nodes which are neither the MDS server, nor one of the OST servers).

The program to be run by the processing components (i.e., the background processes), is specified by every Active Storage job. The program can belong to a generic "library" of Active Storage programs, or can be particular to the parallel application running in the compute nodes. It would also be possible to run several processing components per node, with different programs, and perform multiple data conversions on the output data produced by the remote nodes (the 1R→1W processing pattern, described in Table 1, would be very useful in this case).

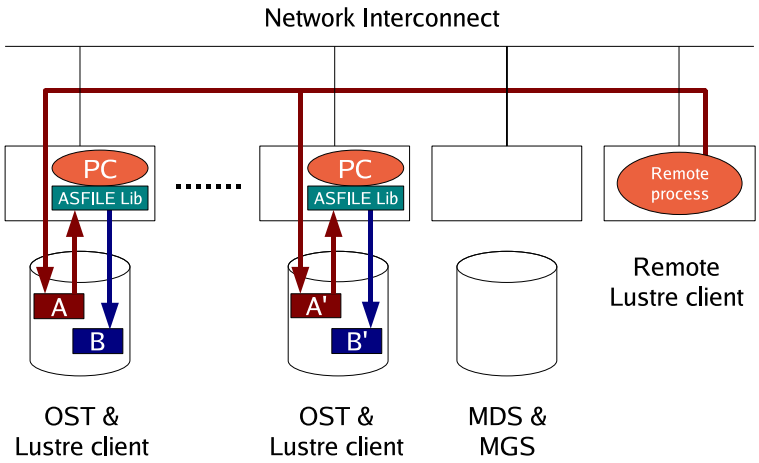


Figure 3: User-space implementation of Active Storage in Lustre

One important point of the user-space approach is the synchronization between the remote clients and the processing components. If the input file in an OST is empty, or if the file pointer has reached the end of the file, the processing component in that OST must wait for more data. On the other hand, if the remote client is finished, and is not going to send more data, the processing component must know that to complete its job and exit.

In order to simplify the implementation of the processing components, we have built a library which contains functions that hide the synchronization details between the remote and local processes. Some of the functions in this library are `asopen`, `asread`, `aseof`, and `asclose`. These functions have the same interface as the corresponding `fopen`, `fread`, `feof`, and `fclose` counterparts, and behave in the same way. These functions constitute what we call the *ASFILE interface*. Although this library creates a new API that the programmers must know and use, it is easy to get familiar with it because it is pretty similar to the existing FILE interface.

In both the kernel- and user-space implementations of Active Storage, we must be able to select the OST where to store every file used by the processing components. This is because each pair of input and output files should be located in the same OST. The Lustre `lfs` command (with the `setstripe` option), and the `llapi_file_create` function of the Lustre API, fulfill this requirement. Other parallel file systems, however, lack this function, and should be enhanced to properly deploy Active Storage on them. This is the case of PVFS [4] [1].

The user-space approach has some important usability advantages over the kernel-space implementation of Active Storage. First, the development of application code for the AS processing components in user-space is much easier than in kernel-space, where debugging, for example, is very difficult for most scientific application programmers. Second, porting from one parallel file system to another is greatly simplified, as long as the target file system has some basic features (like the ability of specifying the layout of a file, as we have mentioned above). And third, the processing components have an overview of the entire file system; therefore, it is easy to develop new processing patterns, to access other files in the file system (i.e., configuration files), to discover the layout of a file across the OST's (this information can be used by a processing component to know which portions of a striped file are stored in its corresponding OST), etc.

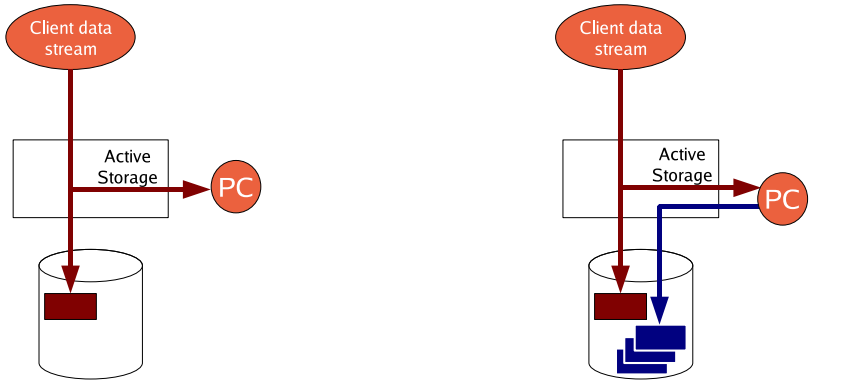
Our current user-space implementation, as showed in Figure 3, has an additional advantage over the kernel one. The advantage is that the processing components are not in the data paths from the remote clients to the OST's, so they do not become a bottleneck. In the kernel-space implementation, however, if the clients send data to the OST's faster than the processing components can treat them, they will have to wait once the main memory of the corresponding OST's gets full (remember that the Active Storage module has to make a copy of the data written by the remote clients before letting it go forward to the disk).

As we can see, all the above makes our proposed user-space approach more flexible, portable, and readily deployable than the kernel-space counterpart. For example, it has been possible to port it to the PVFS file system with minor changes (related to the way file layouts are specified). The kernel-space approach had required to modify the server-part components of PVFS. Moreover, the experimental results described in section 5 also show that the user-space approach provides a better performance. This is because it takes advantage of the buffer cache of the operating system, so it is able to overlap CPU and I/O operations more efficiently.

It is important to realize that our approach is different to running a parallel application on the storage nodes in several ways. Firstly, there are not interactions among the processing components (although they could be certainly possible). Secondly, the data traffic to be processed by every processing component is entirely local. And thirdly, the execution is data-driven.

A regular parallel application should be carefully implemented in order to achieve the same properties, and the implementation could be even impossible. A first problem is that there is not always an interconnect which makes the communication between different OST's possible (although, obviously, the OST's must be able to communicate with the remote clients and the MDS/MGS server). A second one is

[1]There exists some preliminary work in this regard, but it is not included in the official branch yet.



(a) Processing pattern 1W→0 (b) Processing pattern 1W→#W

Figure 4: Two possible processing patterns in Active Storage

Table 1: Processing patterns for active storage in a parallel file system

Pattern	Description
1W → 2W	Data will be written to the original raw file. A new file will be created that will receive the data after it has been sent out to a processing component.
1W → 1W	Data will be processed, then written to the original file
1R → 1W	Data that was previously stored on the OBD can be re-processed into a new file.
1W → 0	Data will be written to the original file, and also passed out to a processing component. There is no return path for data, the processing component will do 'something' with the data.
1R → 0	Data that was previously stored on the OBD is read and sent to a processing component. There is no return path.
1W → #W	Data is read from one file and processed, but there may be many files that are output from the processing component.
#W → 1W	There are many inputs from various files being written as outputs from the processing component.

that the application should be aware of the file distribution. The number of files per OST can change during the course of the time, and files can be unevenly distributed. Therefore, the number of "processing components" of the parallel application per OST's should be variable, and could change from run to run, or even during the same run. And a third problem is that users do not usually have permission to access the storage nodes. Hence, they can not run any application on them.

4.2 Striped Files

Many applications split files across several OST's in order to profit the aggregate bandwidth provided by the storage nodes. Although the specific treatment of the striped files by Active Storage is out the scope of this paper, we would like to comment some possible approaches.

In the user-space implementation of Active Storage, the processing components see all the files stored in the parallel file system. Moreover, if a processing component can read a file, it can read it entirely, either if the file is striped or not.

To process a striped file, Active Storage launches a processing component per each OST used by the file, and every processing component processes the file chunks stored in its OST. If the file records are not chunk-aligned, each processing component processes only those records which start in one of its local chunks. In the latter case, the I/O operations will be "almost" entirely local.

With respect to the visibility of the chunks, there are two options: the processing components can see all the chunks of a file, or they can see their local chunks as a single, contiguous file. The latter can be transparently provided by Active Storage by using a solution like that proposed by Mitra *et al.* [19] for their non-intrusive, log-based I/O mechanism. This transparent approach also allows Active Storage to use programs which know nothing about striped files as processing components.

In the kernel implementation, the Active Storage module only deals with objects (there is an object per file which stores all the file's chunks in the corresponding OST). It does not see whole files or other files. Therefore, the module (and, in turn, the processing components) can not access to chunks stored in other OST's. This restriction also limits the applicability of the kernel implementation to striped files.

4.3 Processing Patterns

In the examples above, we have showed the basic operation of the processing pattern supported by both our current kernel- and user-space implementations. We refer to this pattern as 1W→2W. However, other types have been conceptualized, and will be eventually available in upcoming releases. Figure 4 graphically shows two processing patterns, and Table 1 shows a few more. The symbolic description uses W for writes, R for reads, and numbers, or the # sign, for more that one data stream. The left side of the symbolic name specifies input, and where it originates. Inputs can be read from disk (R), or be copied from a data stream being written (W). Outputs can be to the disk (W), or as a data stream being sent to a reading process (R).

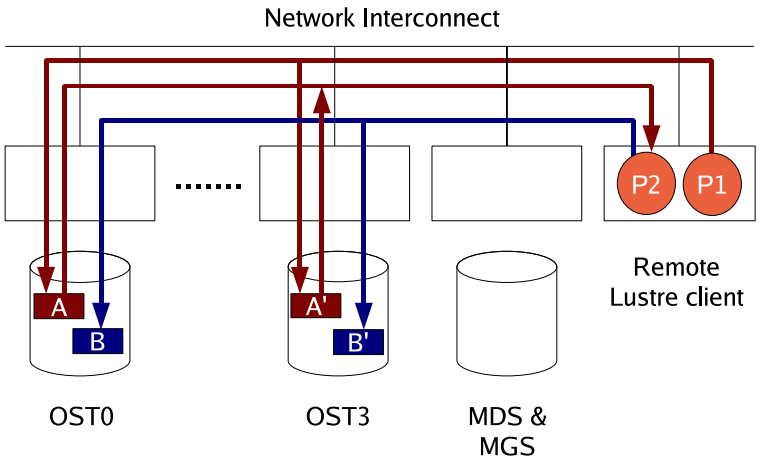


Figure 5: Lustre configuration without Active Storage. Both output files (A and B) are striped across all the OSTs with a stripe size of 128 KB.

Table 2: Technical features of every node in our system under test.

CPU	2 x Itanium 2 at 1 Ghz
RAM	6 GB
NIC	Dual-Link Myrinet NIC (M3F2-PCIXE-2)
Hard disk (OS)	HP MAN3367MC (36.40 GB)
Hard disk (Lustre)	MAXTOR ATLAS10K4_73SCA (73.56 GB)
OS	Linux 2.6.9
Lustre	1.5.95 (1.6beta5)

The above processing patterns suppose that the information is shipped from the remote clients to the processing components. If the information must travel in the opposite direction, and we do not want to (or can not) modify the code of the remote client, then the solution is a little more complex. This solution is out of the scope of the present paper, and remains as future work.

5. EXPERIMENTAL RESULTS

In order to prove the advantages of our Active Storage proposal in user-space for parallel file systems, we have compared our kernel- and user-space implementations with the Lustre configuration depicted in Figure 5, which does not use Active Storage at all. As we can see, our system under test has 1 compute node, 1 MDS/MGS server, and 1, 2, or 4 OST's. All nodes have the same hardware and software elements. Table 2 shows the technical features of every node.

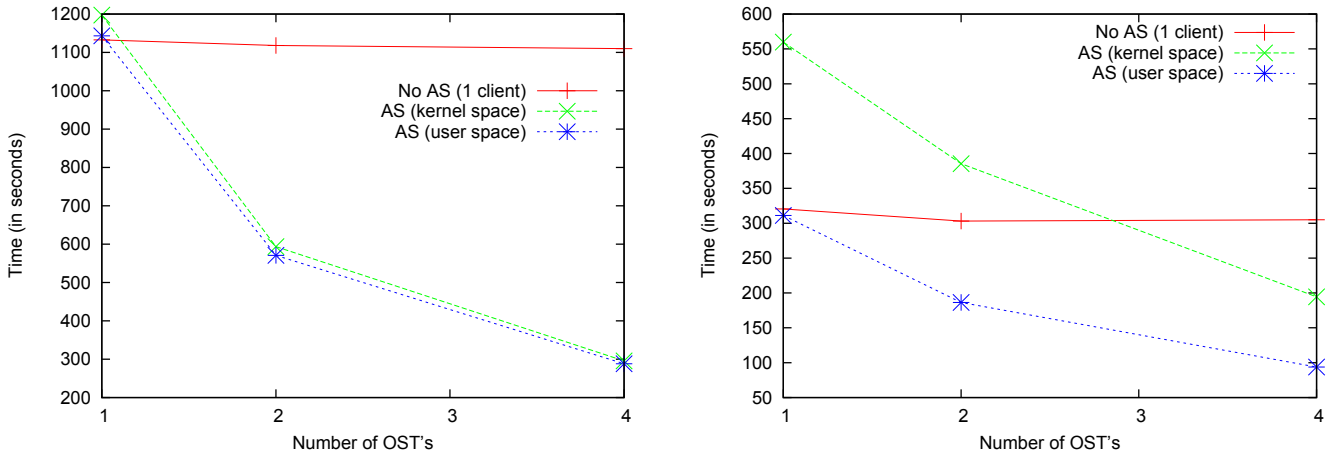
Without Active Storage, a process P1 in the compute node (*remote client*) produces a file of "unprocessed" data which is stored in the Lustre file system (in our case, this process is a `cp` or `dd` command which copies an existing file to the parallel file system). This file of unprocessed data is the "input" file of a second process P2 which, after some processing, produces an "output" file with processed data.

With Active Storage, the process P2 becomes our "processing component", and an instance is run on every OST. Now, the process P1 is a shell script which remotely launches the processing components, then copies the files to be processed to the storage nodes, and finally waits on the processing components to finish.

We have used a microbenchmark that we call DSCAL, and an application called AMINOGEN, to perform the processing job. The former reads an input file which contains doubles, one per line, and multiply every number by a scalar which is passed as a program argument. The resulting doubles are written to the output file, one per line. The latter is a proteomics application aimed at enhancing the process of identifying amino acid polymers (proteins) in experimental samples obtained at the Mass Spectrometry Facility of the Environmental Molecular Sciences Laboratory (EMSL) at PNNL. Mass-spectroscopy based proteomics produces a very accurate measurement of protein fragments based on their mass-to-charge ratio (m/z). For each m/z peak in an experimental spectrum, one needs the list of corresponding candidate sequences having the correct mass. AMINOGEN produces this list for each spectral peak using its m/z value, its range of uncertainty, and knowledge of the masses of each amino acid residue. The m/z and tolerance pairs are in the input file, while the resulting amino acid sequences, along with their m/z values, are stored in the corresponding output file.

For DSCAL, the input file is 1 GB in size, and contains almost 121 million doubles stored as strings, one per line. This application produces an output file whose size is around 1.7 GB. In the AMINOGEN case, the input file contains 4 pairs of mass and tolerance which add up to 44 bytes. The generated output file, however, has a size of 14.2 GB. Without Active Storage, both input and output files are striped across all the OST's by using a stripe size of 128 KB. With Active Storage, every file is split into as many subfiles as available OST's, all of them having roughly the same size.

In the next subsections, we present the results obtained. Every number is the average of 3 runs. The standard deviations are small, and most of them are smaller than 1% of the mean. Both DSCAL and AMINOGEN have been run for the three aforementioned configurations of Lustre: without Active Storage, with the kernel-space implementation



(a) DSCAL

(b) AMINOGEN

Figure 6: Overall execution time for DSCAL and AMINOGEN.

of Active Storage, and with the user-space one. We have also used 1, 2, and 4 OST's in order to show the potential scalability of our proposal.

We are aware that our Lustre configuration without Active Storage only uses one node to perform the processing tasks, while the other two Active Storage configurations use up to four node to do the same work. Therefore, the latter can obviously be faster than the former. However, what our configurations prove is that there are cases where the work to do can be easily parallelized by Active Storage, which leverages the processing power provided by the storage nodes, and reduce the network traffic and I/O latency and the same time, what improves the overall system performance.

On the other hand, it is also worth to note that cases like that described by the configuration without Active Storage are not uncommon. For example, these cases arise when a set of files must be compressed before storing them in a tertiary storage system, or when some modifications must be performed on hundreds or thousands of existing files [2].

5.1 Overall execution time

Figure 6 shows the overall execution time for every benchmark and configuration. This is the time to carry out the entire job, either in the client or in the OST's, and it also includes the time taken to transfer data from the client to the OST's.

In the DSCAL application, the completion time for the three configurations of Lustre is roughly the same when there is 1 OST. When there are 2 and 4 OST's, the completion time is respectively divided by 2 and 4 for the two Active Storage implementations, and basically remains the same with the *No Active Storage* configuration (actually, the time in this case decreases a little as the number of OST's increases, because the input and output files are striped, and the aggregate I/O bandwidth is bigger). Therefore, these results clearly show the scalability that we can expect to achieve with Active Storage.

[2]The netCDF Operator Homepage, `http://nco.sourceforge.net`, provides a good example of programs to perform operations on existing netCDF files.

In the AMINOGEN case, we can say the same about the results obtained by two configurations: without Active Storage, and with the user-space implementation of Active Storage. That is, the execution time is quite similar when there is 1 OST (although it is smaller with Active Storage because it does not have to transfer 14.7 GB across the network), and is roughly divided by 2 and 4 for the Active Storage configuration when there are respectively 2 and 4 OST's.

The results for the kernel-space implementation of Active Storage, however, are quite different. In this case, the overall execution time is 79.9%, 106.8%, and 107.4% greater than that obtained by the user-space counterpart when there are 1, 2, and 4 OST's, respectively.

The main reason for this poor performance is that the kernel-space approach does not use the available buffer cache of the storage node, nor does it implement its own cache (however, Lustre clients, such as our user-space approach, use the buffer cache provided by the operating system). By not using the buffer cache, the overlap of CPU and I/O operations is very small (or it does not exist at all). This represents a serious performance degradation in high I/O workloads like AMINOGEN. Also note that implementing a buffering mechanism inside the processing components only helps to reduce the number of system calls, but it does provide an overlap of CPU and I/O operations by itself.

Another reason for the low performance of the kernel implementation is that the I/O operations are carried out by only one kernel thread, whereas the user-space counterpart takes advantage of the different I/O kernel threads used by Lustre in the storage nodes.

5.2 Completion time in the client

Another interesting result is the completion time in the client, i.e., the time that the client takes part in the job and, therefore, the time that it must wait before doing other things. Tables 3 and 4 show these times for DSCAL and AMINOGEN.

Without Active Storage, the completion time in the client is equal to the overall execution time, because the client must carry out the computational job. However, with Ac-

Table 3: Completion time in the client (in seconds) for DSCAL.

	1 OST	2 OSTs	4 OSTs
NOAS	1132.63	1117.90	1109.97
AS (kernel space)	29.64	12.96	6.17
AS (user space)	30.61	13.39	6.29

Table 4: Completion time in the client (in seconds) for AMINOGEN.

	1 OST	2 OSTs	4 OSTs
NOAS	320.53	303.12	304.90
AS (kernel space)	<1	<1	<1
AS (user space)	<1	<1	<1

tive Storage, the completion time is only the time required to transfer the input file data from the client to the storage nodes, since the job is performed by the processing components in the OST's.

For DSCAL, the completion time in the client decreases as the number of OST's increases. The input file is split into as many portions as OST's, and all portions are transferred in parallel. Therefore, the more OST's, the higher the bandwidth to transfer data from the client to the OST's (the input file is cached in the main memory of the client before every run, so the transfer time does not include the time required to read the input file from disk, which is invariable with the number of OST's).

For AMINOGEN, the completion time in the client is the time to transfer 44 bytes or less, and is very small (less than 1 second).

These results highlight another important advantage of Active Storage: as work is offloaded to the OST's, the remote clients can be assigned to other tasks, and the entire computation can proceed faster.

5.3 Network traffic

The third analysis that we have carried out is the network traffic during the run of every benchmark. We have analyzed the network traffic in the client and the OST's, although we will only show here the amount of bytes sent and received in total in the client NIC, because it represents quite well the overall network traffic. This information is displayed in Tables 5 and 6.

With Active Storage, the network traffic is mainly due to the transfer of the input file from the client to the OST's. In the DSCAL application case, the input file is 1 GB in size. The final amount of bytes sent and received is greater than 1 GB because of the overhead of the Myrinet and Lustre protocols, and because our interconnect is shared with other nodes in the cluster (there are 25 nodes altogether). In the AMINOGEN case, the input file has only 44 bytes, so the network traffic is very small, and it is mainly due to the overhead incurred by the protocols and to unrelated network traffic (note that the network traffic decreases with the number of OST's because the overall execution time also decreases; this also explains why the network traffic is lower in the user-space implementation than in the kernel one, since the former is faster than the latter).

Without Active Storage, the network traffic is much higher due to the fact that both the input and output files must be

Table 5: Network traffic (in MB) for DSCAL.

	1 OST	2 OSTs	4 OSTs
NOAS	2837.03	2840.03	2839.44
AS (kernel space)	1086.12	1063.33	1050.82
AS (user space)	1084.10	1062.43	1050.09

Table 6: Network traffic (in MB) for AMINOGEN.

	1 OST	2 OSTs	4 OSTs
NOAS	14778.00	14808.06	14847.85
AS (kernel space)	22.28	15.77	7.20
AS (user space)	12.12	7.69	4.02

sent across the network. This is especially true for AMINOGEN, where the total amount of bytes sent and received is almost 15 GB. Note that the network traffic does not include the read of the input file to be processed by the client from the OST's. This is obvious for DSCAL, where the total amount of bytes transferred is much less than 3789 MB, which is the sum of one write (1 GB) and read (1 GB) of the input file, and one write (1.7 GB) of the output file. The reason is that the compute node has enough RAM to cache the whole input file. Therefore, with less RAM or bigger files, the network traffic could be higher.

In our system under test, the interconnect does not become a bottleneck because the number of nodes is very small (up to 6, including the MDS/MGS server). However, in a production cluster, with thousands of nodes, and files of hundreds of gigabytes, the situation could be different. In this case, Active Storage could make a big difference because it can substantially reduce the network traffic.

6. CONCLUSIONS

Despite the impressive progress of computer technology in processor, network and storage areas, efficient management of high volumes of data remains to be a challenging problem in many high-performance computing centers. By taking advantage of the underutilized CPU time of the storage nodes in modern parallel filesystems, Active Storage hold a promise of helping to address the challenge for some applications that perform simple processing of the data stored in parallel filesystems such as Lustre or PVFS. The current paper described a new implementation of the Active Storage concept based on a pure user-space approach, and compared it to the existing kernel-space implementation.

The experimental results obtained prove that both implementations are able to take advantage of the extra computing power provided by the storage nodes, scaling up the performance of data-intensive applications, and significantly reduce the overall network traffic at the same time, what can prevent a network bottleneck. For example, for a bioinformatics application example, running on a Linux cluster with Myrinet and using four storage servers, the execution time was reduced by 51.8% over the kernel space approach, and 69.3% over the version of the code that did not use Active Storage.

In addition to offering competitive performance, the user-space implementation appears to be more flexible, portable, and readily deployable than the kernel-space counterpart. For example, after developing this implementation for Lus-

tre, we were able to adopt it with relatively small effort to work with PVFS. We are extending the user-space approach to provide a friendly environment for programmers, to add more processing patterns, and to easily support file striping across several storage nodes.

7. ACKNOWLEDGMENTS

The research described in this paper was supported by the Department of Energy, Office of Advanced Scientific Computing Research at the Pacific Northwest National Laboratory, a multiprogram national laboratory operated by Battelle for the U.S. Department of Energy under Contract DE-AC06-76RL01830. We would also like to thank the anonymous reviewers for the feedback on this papers, and Chris Oehmen and Tim Carlson for the support provided at PNNL.

8. REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *In Proc. of the ACM ASPLOS Conference*, pages 81–91, October 1998.

[2] P. J. Braam, R. Brightwell, and P. Schwan. Portals and networking for the lustre file system. *In Proc. of IEEE Intern. Conf. on Cluster Computing*, 2002.

[3] P. J. Braam and R. Zahir. Lustre technical project summar. *Attachment A to RFP B514193 Response*, July 2001.

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for linux clusters. *In Proc. of 4th Annual Linux Showcase and Conference*, October 2000.

[5] Cluster File Systems Inc. Lustre: A scalable, high-performance file system. *Available at* `www.lustre.org`, 2002.

[6] D. J. DeWitt and P. Hawthorn. A performance evaluation of database machine architectures. *In Proc. of the Int. Conf. Very Large Data Bases (VLDB)*, pages 199–214, September 1981.

[7] D. H. Du. Intelligent storage for information retrieval. *In Proc. of the International Conference on Next Generation Web Services Practices (NWeSP'05)*, pages 214–220, 2005.

[8] EMC. Centera. `http://www.emc.com/products/systems/centera.jsp`, 2007.

[9] E. J. Felix, K. Fox, K. Regimbal, and J. Nieplocha. Active storage processing in a parallel file system. *In Proc. of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, April 2006.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *In Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, October 2003.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. *In Proc. of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–284, June 1997.

[12] IBM. Storage Tank. `http://www.almaden.ibm.com/StorageSystems/projects/storagetank`, 2007.

[13] IBM Corp. General parallel file system. `http://www.almaden.ibm.com/StorageSystems/projects/gpfs`, 2007.

[14] INCITS Technical Committee T10. Scsi object-based storage device commands (OSD). working draft, revision 10. *Available at* `http://www.t10.org/ftp/t10/drafts/osd/osd-r10.pdf`, July 2004.

[15] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *In SIGMOD Record*, 24(7):42–52, September 1998.

[16] P. M. Kogge, J. B. Brockman, T. Sterling, and G. Gao. Processing in memory: Chips to petaflops. *In Workshop on Mixing Logic and DRAM: Chips that Compute and Remember (at ISCA'97)*, June 1997.

[17] H. Lim, V. Kapoor, C. Wighe, and D. H. Du. Active disk file system: A distributed, scalable file system. *In Proc. of the 18th IEEE Symposium on Mass Storage Systems and Technologies, San Diego*, pages 101–115, April 2001.

[18] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, August 2005.

[19] S. Mitra, R. R. Sinha, and M. Winslett. An efficient, nonintrusive, log-based I/O mechanism for scientific simulations on clusters. *In Proc. of IEEE International Conference on Cluster Computing*, 2005.

[20] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener. Lightweight I/O for scientific applications. *Sandia National Laboratories. Technical Report SAND2006-3057*, May 2006.

[21] Panasas. The panasas activescale file system (PanFS). `http://www.panasas.com/panfs.html`, 2007.

[22] Red Hat Inc. Global file system. `http://www.redhat.com/software/rha/gfs`, 2007.

[23] E. Riedel. Object based storage (OSD) architecture and systems. *Available at* `http://www.snia.org/education/tutorials/2006/fall/storage/Object-based_Storage.pdf`, October 2006.

[24] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. *In Proc. of the 24th Int. Conf. Very Large Data Bases (VLDB)*, pages 62–73, 1998.

[25] S. W. Schlosser and S. Iren. Database storage management with object-based storage devices. *In Proc. of the First International Workshop on Data Management on New Hardware (DaMoN)*, June 2005.

[26] Seagate. The advantages of object-based storage — secure, scalable, dynamic storage devices. *Available at* `www.seagate.com/docs/pdf/whitepaper/tp_536.pdf`, April 2005.

[27] SGI. Infinitestorage shared filesystem cxfs. `http://www.sgi.com/products/storage/tech/file_systems.html`, 2007.

[28] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. *In Proc. of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, November 2006.