Active Storage Processing in a Parallel File System

Evan J. Felix, Kevin Fox, Kevin Regimbal, Jarek Nieplocha

W.R. Wiley Environmental Molecular Sciences Laboratory
Pacific Northwest National Laboratory,
P.O. Box 999, Richland, WA 99352
Tel: (509) 376-1491
Fax: (509) 376-0420
evan.felix@pnl.gov
http://mscf.emsl.pnl.gov

Abstract : *This paper proposes an extension of the traditional active disk concept by applying it to parallel file systems deployed in modern clusters. Utilizing processing power of the disk controller CPU for processing of data stored on the disk has been proposed in the previous decade. We have extended and deployed this idea in context of storage servers of a parallel file system, where substantial performance benefits can be realized by eliminating the overhead of data movement across the network. In particular, the proposed approach has been implemented and tested in context of Lustre parallel file system used in production Linux clusters at PNNL. Furthermore, our approach allows active storage application code to take advantage of modern multipurpose operating Linux rather than a restricted custom OS used in the previous work. Initial experience with processing very large volume of bioinformatics data validate our approach and demonstrate the potential value of the proposed concept.*

1. Introduction

The volume of data has grown exponentially, both in technical computing as well as in the commercial sector. This data explosion has in turn driven the rapid evolution of diverse storage technologies to address needs of different market segments. It is now possible to store terabytes of data with just a few disk drives. Hitachi is shipping 400 GB disks, and will soon release 500GB disks. The software and hardware technology to aggregate and support large file systems also been advancing in the last decade. Proprietary parallel file systems, such as IBM's GPFS [1], Panasas ActiveScale File System [2], SGI's CXFS [3] and many others provide this capability. There are also a number of open source Parallel file systems such as PVFS [4], RedHat (Sistina) GFS [5], and Lustre [6].

As the quantity of data has increased, the need for the ability to process and analyze the data has increased as well. Linux clusters and large Symmetric Multiprocessor

(SMP) computers have been developed to handle the computational loads of modern applications. However, for many data intensive applications, the storage and analysis of data remains as a serious bottleneck. We believe that unexplored alternatives exist to address the problem on the storage side, in particular the parallel file system side, and have been pursuing one of them in the current effort. Parallel file systems use one or more storage servers of some type. Because of the fundamental differences between the historical rates of improvement for the different hardware components in particular CPU and disk, the storage servers have underused processing power. In many cases, the parallel file system server nodes are very similar or identical to the compute nodes deployed in a cluster, and in many cases offer Giga-op/s of processing power.

The original research efforts on *active storage* and *active disks* [7, 8, 9] were based on a premise that modern storage architectures have progressed to a point that there is the real possibility of utilizing that unused *processing power of the drive controller* itself. However, for numerous reasons commodity disk vendors have not offered the required software support and interfaces to make *active disks* widely used. Our current project is developing active storage processing capability in parallel file systems. In this approach, active storage is a system that takes advantage of the underutilized CPU time on *file system servers* to process, not simply store, data. Processing data directly on the storage units can give a dramatic performance increase, by avoiding redundant transfers the data over networks, interconnects, or storage busses. The approach we have been currently pursuing uses these concepts by modifying the Lustre source base to enable the Active Storage concept. The Lustre file system is deployed at numerous sites running Linux clusters including national laboratories such as Lawrence Livermore National Laboratory, National Center for Supercomputing Applications, and Pacific Northwest National Laboratory.

Lustre was designed early on to be a modular, layered architecture. This makes it possible to extend the architecture by adding additional layers. For example if module A is normally attached for communication to Layer B (A⇔B), one could attach a new module C to module B, and then attach module A to module C (A⇔C⇔B). The additional layer could implement almost anything at this point, a simple pass-through inspection layer, a security layer, or a layer which makes large changes to the behavior of the file system. We developed a prototype implementation of active storage in parallel file system by defining and adding an appropriate active storage layer to the Lustre file system. Our early experiences involving bioinformatics applications have already shown viability of this approach, which can perform processing on large data streams without moving it across the network.

The paper is organized as follows. Section 2 provides an overview of Lustre. Section 3 describes the proposed approach and its implementation in the context of Lustre. Deployment and application experience are presented in Section 4. The paper is concluded in Section 5.

2. Lustre Overview

The Lustre file system is an open source file system, currently development is led by Cluster File Systems, Inc., with funding support from the U.S. Department of Energy and other industry partners. Lustre is a highly parallel system, utilizing multiple storage

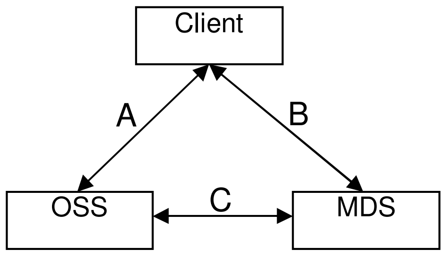

Fig. 1*:* Basic Lustre Components

servers, a meta-data server, and a variety of interconnects to serve thousands of clients at very high throughput rates. Among the supported interconnects are Quadrics, Infiniband, and Ethernet.

The basic structure of Luster is shown in Figure 1. Lustre Meta-Data Servers (MDS) handle the data associated with all files in the system, including the directory tree, filenames, and file permissions. System wide file location information is also stored on the MDS; this information is needed for the client nodes to perform I/O operations on the files. Lustre *Object Storage Servers* (OSS) are responsible for serving data. They support a simple object based protocol to clients so that data can be moved quickly from multiple OSS nodes to client nodes. High throughput numbers are achievable by using many OSS nodes in parallel. The Lustre file system is very scalable. The addition of more OSS nodes can improve the performance, and size of the system. Since true *Object Based Disks* (OBD) are not readily available in hardware, the Lustre code base places a 'filter' layer over the Linux *ext3* journal file system, which provides a robust failure recovery mechanism.

Lustre client nodes communicate with the MDS and OSS nodes, and present a single coherent file system to the user. The client communicates with the MDS server in order to perform the needed creation, update, or deletion of files and directories. Once files are in place, the client sends and receives portions of files directly from the OSS

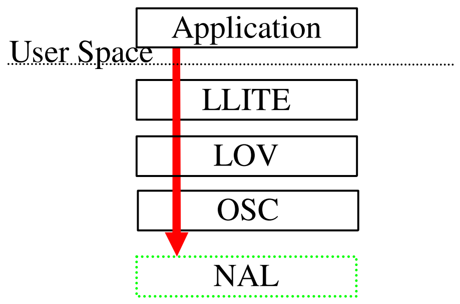

Fig. 2: Lustre Client Modules

nodes. Files can be striped across many OSS nodes so that successive blocks of a single file are stored on many servers.

Lustre uses a variant of the Portals networking layer that was originally developed at Sandia National Laboratories [10]. Additions were made to the base Portals source code to facilitate extensions needed by the Lustre system. One of these changes allows portals traffic to 'route' between multiple interconnect types. This infrastructure provides the Network Abstraction Layer needed to facilitate Lustre traffic over the various interconnects. This also allows for additional interconnect types to be included easily without major modifications to the Lustre code base.

Lustre also employs a Distributed Lock Manager (LDLM) that runs on all the nodes, that is based off work done in the VAX Cluster Distributed Lock Manager [6]. This lock manager handles file extent reservations to specific nodes. It helps ensure consistent data semantics, so that multiple clients can be modifying the same file simultaneously without sacrificing data integrity.

3. Technical Approach

Our approach exploits the modular structure of Lustre. The classic Lustre client layers are shown in Figure 2. An application 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) modules, and the Logical Object Volume (LOV). The LOV layer is responsible for the dispatching of the data related calls to the set of Object Storage Clients (OSC) that are available. It performs these duties based on striping information

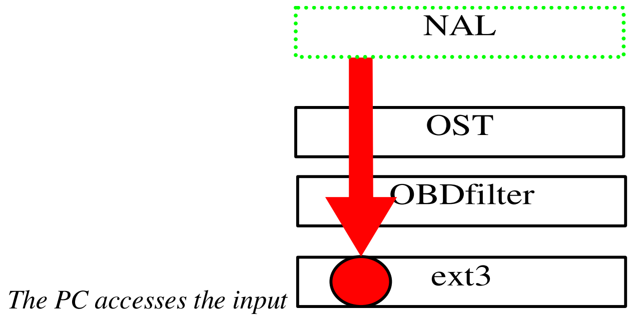

Fig. 3: Lustre OSS Modules

obtained from the Meta Data Server (MDS). The OSC layer packages up the requests and sends them to an OSS node serving the Object Storage Target (OST) that is specified, over the Portals Network Abstraction Layer (NAL).

The Object Storage Servers (OSS) serve a set of OST's, these targets relate directly to the underlying disk file systems that are available, see Figure 3. The OST layer receives the requests from the NAL, and feeds the requests down to the OBD filter layer. This OBD filter wraps 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-though module, until such time as an active storage task is requested. To initiate the active storage process a client application will create empty output 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 4 shows the processing that takes place as the file is written. For example, the client would create two empty files A and B, and then sends 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 made. A small helper process is started that sets up the correct interface, and then starts the PC. Once the processing component completes is work and exits, the helper process cleans up the interface files and then terminates.

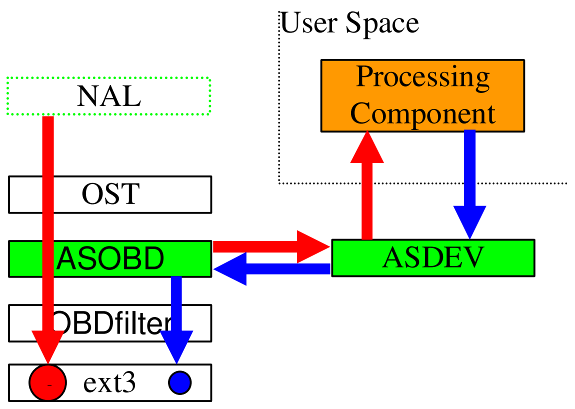

Fig. 4: Active Storage Modules

Once the files have been created, linked, and the processing component is initialized, the client will start writing data to file A, using normal file API calls. The active storage layer sees this data, and passes the data 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 *read*() operation on the device interface it will either get the waiting data, or will block until a new data block becomes available. Once the processing component has completed its work, any output is written out by calling *write*() on the AS device interface. 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 appear in file B.

The processing component accesses the input and output files as streams, which resembles the approach described in [9]. The simple stream model allowed one buffer of the stream to be available at a time. That work relied on using a custom Disk OS that restricts the actions a program can take. In the active storage approach we are pursuing, the processing component has full access to the normal Linux Operating environment, which does not restrict it as much as the proposed DiskOS model described in [9]. Furthermore, for active storage applications many types of processing will become soon available, based on abstracting input files, output files, and disk-resident files as streams.

In order to add a new Active Storage Processing Component a user must work within a specific framework we provide. First a processing component must be added to the database of Processing Components, which also specifies what the arguments to the

PC will be. When the PC starts it will be created from within a special kernel context, and will not have direct access to the standard IO streams. The PC is also given a device file that it can read or write to. Any read type streams will be read from this device in a packetized form, with a block header describing the data block that will be read next. Data written to this device file is then written to the output file. The PC has access to most of the Operating System services such as network, and local file systems.

3.1 Active Storage Processing Patterns

Many patterns of processing can be supported by the active storage system. The basic file write which results in a raw file and a processed file (1W->2W) was used in the example above. Other types have been conceptualized and are under development, see Table 1. 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 bead 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).

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
#W->1W	There are many inputs from various files being written as outputs from the processing component.
1R->1R	Data is read from a file on disk, sent to a processing component, then the output is sent to the reading process.

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

4. Deployment and Application Experience

Our current implementation supports the 1W->2W streaming type. The framework of the AS modules is being extended to support more general input output methods. Once input code is developed to read data directly from the objects on disk, the

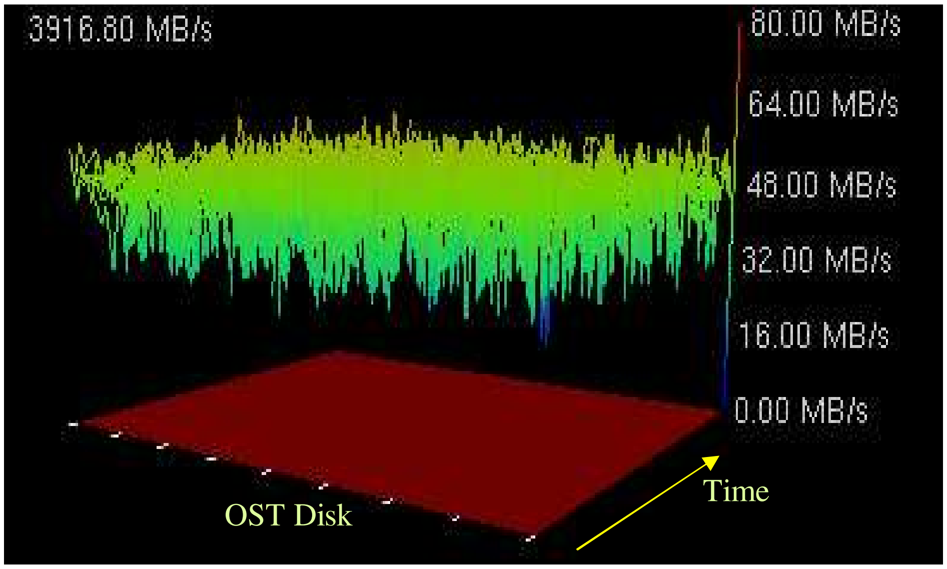

Fig. 5: I/O bandwidth profile in the StorCloud experiment across Lustre servers collected and visualized using NWPerf

1R-> methods will become available. Our initial development and testing work around active storage used a simple *gzip* compression processing component. It that was able to improve the compression time of a data stream, as the raw data only needed to be sent into the file system once. More recently we began deployment of active storage in context of scientific data intensive applications, namely in the area of bioinformatics.

As part of the StorCloud Challenge of SC'2004 conference, we demonstrated an active storage application based a single Lustre file system striped across 40 OSS servers, each containing 24 400GB disk drives configured as four six-disk RAID0 sets, for a total of four OST's per server. Formatted, this presents itself as a 348TB file system to client nodes. A real-time write speed visualization for display on the conference floor is presented in Figure 5. It was based on the NWPerf profiling and visualization tool [11] used to collect system profiling data during the experiment. There data was collected at two second intervals, and there are 64 intervals shown in Figure 5.

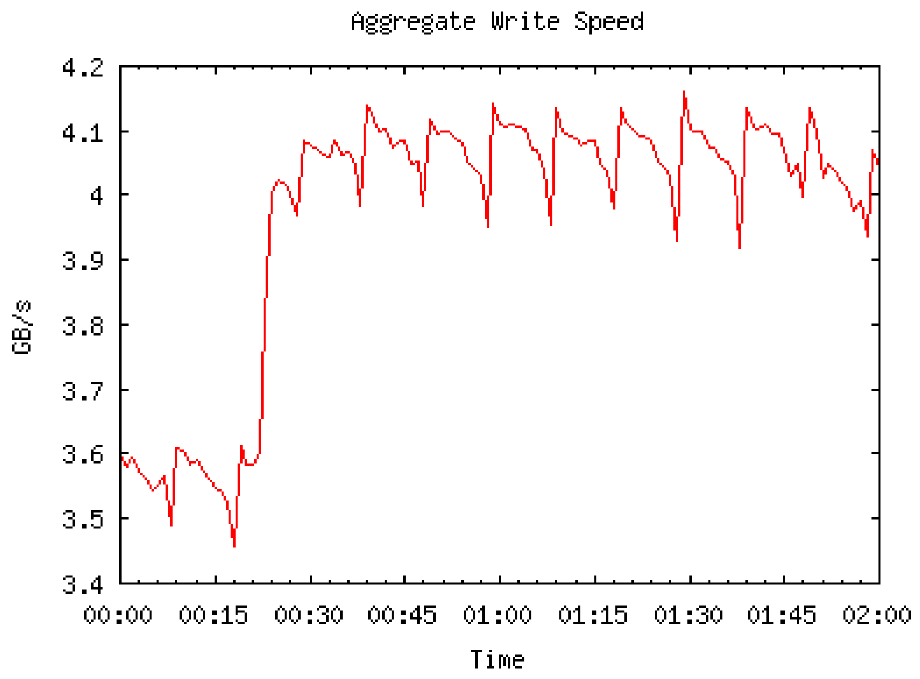

Fig. 6: Aggregate bandwidth in the StorCloud experiment

The Lustre file system used was based on a beta version of Lustre 1.4, and augmented with our active storage module. The application, running within the active storage module, calculated and stored all possible amino acid sequences based on an input mass and tolerance observed experimentally at the high-performance Mass Spectrometry Facility of the Environmental Molecular Sciences Facility (EMSL) at PNNL. For the StorCloud Challenge, a master application running on a single workstation wrote files to the Lustre file system containing a target mass and tolerance. The active storage module read this information, calculated and stored the resulting protein mass and sequence data to a second file within the file system. Since the Active Storage component is running on the OSS where the disks physically reside, very little data crossed the network.

During the challenge period we observed aggregate writes speeds between 3.5GB/s and 4.2GB/s, as seen in Figure 6. This rate was sustainable for many hours as long as there was processing to be done. The raw disk arrays are capable of higher write speed, 8GB/s. In addition to the time taken to run active storage processing application, further investigation using the *oprofile* tools showed that the bottleneck for this application is the user memory to kernel memory data copy call. Some improvement was attained by moving to a different memory copy function that is provided in the kernel. As a part of continuing work on optimizing the active storage module implementation and its integration with Luster framework, we planning to directly map the processing component pages directly to the file system output routines and bypass this memory copy.

Since the interconnect in the experiment was a Gigabit network, at these speeds it is only possible for a single client to write about 80-90 MB/s. The application written can perform at this level and saturates the network. Limitations like this make the Active Storage approach very attractive since it could 1) eliminate the network bandwidth bottleneck and 2) exploit processing power of the servers deployed in the Lustre file system.

5. Conclusions

The concept of active storage/disks was actively investigated during previous decade. The previous work has concentrated primarily on adding processing capabilities directly to available disk processors. The active storage approach that we propose is targeting parallel file systems deployed in Linux clusters instead in order to reduce or even eliminate some data transfers between servers and clients in parallel filesystems deployed in Linux clusters. We developed a prototype implementation by defining and adding an active storage module to the Lustre file system. Our early experiences involving bioinformatics applications have already shown viability of this approach, which can perform processing on large data streams without moving it across the network. We are extending the active storage module to provide other types of processing and working on advancing technology and framework for active storage processing in Lustre.

Acknowledgements

This work is funded by the U.S. Department of Energy, the office of Biological and Environmental Research, Office of Science at the Pacific Northwest National Laboratory (PNNL) under contract DE-AC06-76RLO 1830 with Battelle Memorial Institute and the Computational Sciences and Engineering Initiative at Pacific Northwest National Laboratory. The work was performed in the Molecular Science Computing Facility at the W.R. Wiley Environmental Molecular Sciences Laboratory, PNNL.

References

[1] IBM Corp. General Parallel File System,
http://www1.ibm.com/servers/eserver/clusters/software/gpfs.html
[2] Panasas, http://www.panasas.com/panfs.html
[3] SGI® InfiniteStorage Shared Filesystem CXFS,
http://www.sgi.com/products/storage/tech/file_systems.html

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, ``PVFS: A Parallel File System for Linux Clusters'', Proceedings of the 4th Annual Linux Showcase and Conference, 2000.
[5] Red Hat Global File System, http://www.redhat.com/software/rha/gfs/
[6] Peter J. Braam *et al*, The Lustre Storage Architecture, available at www.lustre.org, 2004.
[7] Erik Riedel, Garth Gibson, Christos Faloutsos, Active Storage for Large-Scale Data Mining and Multimedia, Proc. 24th Int. Conf. Very Large Data Bases, VLDB, 1998.
[8] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein, "*A Case for Intelligent Disks (IDISKs)*," SIGMOD Record, 24(7): 42-52, September 1998.
[9] Anurag Acharya, Mustafa Uysal, Joel Saltz, Active Disks: Programming Model, Algorithms and Evaluation, Proceedings ASPLOS'98. 1998.
[10] Peter J. Braam, Ron Brightwell, Phil Schwan, Portals and Networking for the Lustre File System, Proceedings of IEEE International Conference on Cluster Computing. 2002
[11] R. Mooney, R. Studham, K. Schmidt, J. Nieplocha, NWPerf: A System Wide Performance Monitoring Tool for Large Linux Clusters, Proceedings of the IEEE CLUSTER'04. San Diego. 2004.