Active Storage using Object-Based Devices

Tina Miriam John Anuradharthi Thiruvenkata Ramani John A. Chandy
Department of Electrical and Computer Engineering
University of Connecticut
Storrs, CT USA
{tina.john,anuradharthi.thiruvenkata_ramani,john.chandy}@uconn.edu

***Abstract*—The increasing performance and decreasing cost of processors and memory are causing system intelligence to move from the CPU to peripherals such as disk drives. Storage system designers are using this trend toward excessive computation capability to perform more complex processing and optimizations directly inside the storage devices. Such kind of optimizations have been performed only at low levels of the storage protocol. Another factor to consider is the current trends in storage density, mechanics, and electronics, which are eliminating the bottleneck encountered while moving data off the media, and putting pressure on interconnects and host processors to move data more efficiently. Previous work on active storage has taken advantage of the extra processing power on individual disk drives to run application-level code. This idea of moving portions of an application's processing to run directly at disk drives can dramatically reduce data traffic and take advantage of the parallel storage already present in large systems today. This paper aims at demonstrating active storage on an iSCSI OSD standards-based object oriented framework.**

I. Introduction

The cost and silicon real estate needed for any particular computational need is continually dropping. At some point, additional processing power can be had at negligible cost. The question then becomes simply where to place this computation power in a system to support the widest range of tasks efficiently. The contention of this work is that processing power is already moving into peripheral devices and that applications can achieve significant performance gains by taking advantage of this trend. Specifically, this work focuses on how data-intensive applications can directly exploit the processing power of the controllers in individual commodity disk drives to improve both individual application performance and system scalability.

The same trends in chip technology that are driving microprocessors toward ever larger gate counts drive disk manufacturers to reduce cost and chip count in their devices while simultaneously increasing the total amount of local processing power available on each device. One use for this increasing computation power on disk controllers is to enrich their existing interface. For example, recent advances in network-attached storage are integrating storage devices into general-purpose networks and offloading a range of high level functions directly to the devices. This eliminates servers as a bottleneck for data transfers between disks and clients and promises significant improved scalability through higher-level interfaces. At the same time, as systems get faster and cheaper, people compute on larger and larger data sets. A large server system today will easily have a hundred disk drives attached to it. This large number of drives is necessary either to provide sufficient capacity or sufficient aggregate throughput for the target application. Taking this trend and extrapolating to future drive capabilities gives a promising picture for on-drive processing.

A pessimistic value for the on-drive processing already in today's commodity SCSI disk controllers is 400 MHz, with perhaps 100 MB/s of sustained bandwidth in sequential access. This means that a system with one hundred disks has 40 GHz of aggregate processing power and 10 GB/s of aggregate bandwidth at the disks. Typical multiprocessor systems can not achieve these aggregate values in a single node, but computing clusters can be constructed to achieve the same capability. The advantage of the active disk cluster over a compute node cluster, is the ability of the disk to manage the data directly on disk in terms of layout and scheduling. Data processing at the disk can also reduce the amount of data transferred, thereby reducing communication costs.

In addition, as storage is connected to a large collection of hosts by taking advantage of network-attachment and storage area networks, the interconnection network will rapidly become a principal bottleneck in large-scale applications. If data can be processed directly by the devices at the "edges" of the network, then the amount of data that must be transferred across this bottleneck can be significantly reduced.

Recent studies show that the storage demands are also growing rapidly and if this trend continues, storage administration costs will be higher than the cost of the storage systems themselves. Therefore intelligent, self managing and application aware storage systems are required to handle this unprecedented increase in the storage demands. To be self managing, the storage device needs to be more aware of the data it is storing. But the traditional block interface to storage systems is very narrow and cannot convey any such additional semantics to the storage. This forms the fundamental motivation behind revamping the storage interface from a narrow, rigid interface to a more "expressive" and extensible interface. This new storage interface is termed as the object based storage interface.

Storage devices that are based on this object based interface (Referred to as Object Based storage devices) will store and manage data containers called objects which can be viewed as a convergence of two technologies: files and blocks [1]. Files have associated attributes which convey some information

about the data that is stored within. Blocks, on the other hand, enable fast, scalable and direct access to shared data. Objects can provide both the above advantages. The NASD project at CMU [2], [3] provided the initial thrust for the case of object based storage devices. Object based storage or similar techniques has been apparent over the last decade in many parallel file systems decade including PVFS [4], Lustre [5] and Panasas [6]. However, these these high-performance storage system implementations use proprietary interfaces and hence limit interoperability. Object-based storage has been formalized as a T10 SCSI standard [7] but has yet to see adoption in any widely available commercial device.

II. Background and Motivation

The database machines proposed and developed in the 70s and 80s are the main relatives of Active Storage systems. They had special processing components that handled the execution of portions of the function of a database system. These devices were custom-designed to implement a particular algorithm and assumed a particular set of queries and a particular data layout. There are several classes of machines, with differing levels of complexity. The prevailing counter-arguments to the database machines of the 80s were that 1) for a significant fraction of database operations, such as sorts and joins, simple select filters in hardware did not provide significant benefits, 2) special-purpose hardware increased the design time and cost of the machine, and 3) a single general purpose host processor was sufficient to execute select at the full data rate of a single disk [8], [9]. Commercial data warehousing systems from DATAllegro and Netezza have shown the promise of these systems but they have been limited to database processing.

Previous work on active disks was brought to life by the very database machines described above. They provide an environment for executing application code directly at individual drives [10]. By partitioning processing across hosts and storage devices, it is possible to exploit the cycles available at storage, reduce the load on the interconnection network, and perform more efficient scheduling. For example, an application that applies selective filters to the stored data and only ships summary information across the network, or that makes scheduling decisions based on local information at the individual drives can make more effective use of network and host resources. This promises both improved individual application performance and more scalable systems.

The goal of this paper is to demonstrate an object oriented framework for supporting applications that benefit from an Active Storage system, such as the Active Disks mentioned above. The framework can support a variety of data-intensive applications including database operations. In the following sections, we describe the basic approach that was followed to come up with the framework. Then we go on to describe the implementation details of the sample application developed, to test out the framework. Finally we describe our experimental setup and compare the performance of an existing single server system to a prototype system emulating Object Oriented Active Disks.

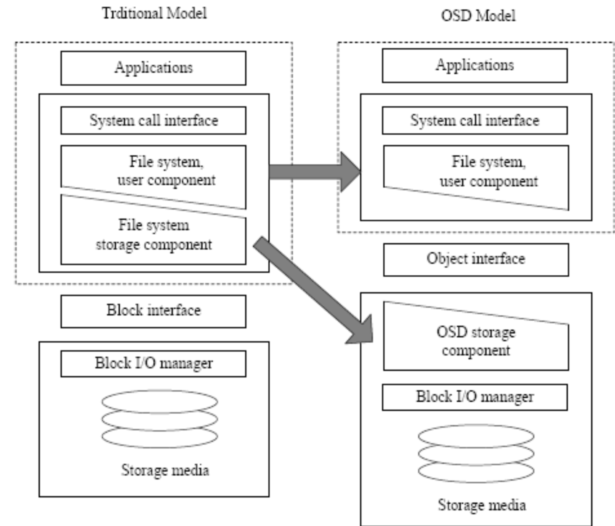

Fig. 1. Comparison of traditional and OSD storage models

III. Overview of Object-Based Storage Devices

The Object-Based Storage Devices (OSD) specification [7] has introduced a new set of device-type specific commands into the SCSI standards family. The specification defines the OSD model and their required commands and command behavior.

Figure 1 illustrates the basic model of OSD in comparison to a traditional block-based device model for a file system. The traditional functionality of file systems is divided mainly so that the increased intelligence that is available in such storage devices can be taken advantage of. Object-based Storage devices have the capability of managing their storage capacity and shows file-like storage "objects" to their hosts. These objects behave exactly like files. They can be created and destroyed and can grow and shrink their size during their lifetimes. As in a file system, commands are available that can be used to read or write any consecutive stream of the bytes making up a storage object. The main difference here is that, in addition to storing data in objects, the OSD storage management component also maintains other information about the storage objects in the form of object attributes, e.g. size, usage quotas and associated user name.

A. OSD Objects

Storage objects that are used to store regular data are called user objects. Three other kinds of objects are also defined in the specification, namely, root object, partition objects and collection objects. There is one root object for each OSD logical unit [11]. The root object forms the top most module of an OSD logical unit similar to a partition table for a logical unit of block devices. User objects are placed in partitions that are represented by partition objects. The maximum number of partitions permissible within a logical unit is given by a specific quota defined in the root object. Every user object belongs

Partition_ID	User_Object_ID	Object type
0	0	root object
2^{20} to $2^{64}-1$	0	partition object
2^{20} to $2^{64}-1$	2^{20} to $2^{64}-1$	collection/user object

TABLE I
OBJECT IDENTIFICATION NUMBERS

to one and only one partition. Collections are represented by collection objects. Each collection object belongs to one and only one partition and may contain zero or more user objects belonging to the same partition. Unlike user objects, the other three kinds of objects do not contain a read/write data area. All relationships between objects are represented by object attributes discussed in the next section.

The unique combination of the Partition ID and the User Object ID identifies each OSD object as illustrated in Table I. The ranges not specified in the table are reserved.

B. Object Attributes and Attribute Pages

Object attributes relates meta data with any OSD object, i.e. root, partition, collection or user. They are organized into pages for identification and reference. Each attribute page associated with an object has a unique attribute page number ranging from 0x0h to 0xFFFFFFFEh. Page numbers from 0x0 to 0x2FFFFFFF can only be associated with user objects. Each attribute in an attribute page has an attribute number between 0x0 and 0xFFFFFFFE that is unique within the attribute page. The last attribute number, i.e. 0xFFFFFFFF, is used to represent all attributes within the page when retrieving attributes.

C. Commands

The OSD commands execution follows a request-response model as defined in SCSI Architecture Model (SAM-3) [11]. This model can be represented as a procedure call as following:

The OSD commands use the variable length Command Descriptor Block(CDB) format defined in SPC-3 but have a fixed length of 200 bytes. An opcode 0x7F in CDB differentiates the OSD command from commands of other command sets. The service action field in the CDB specifies one of the twenty-three OSD service requests defined in the OSD specification. Every CDB has a Partition ID and a User Object ID, to identify the requested object in a logical unit. Any OSD command may retrieve attributes and any OSD command may store attributes. Twenty-eight bytes in CDB are used to define the attributes to be set and retrieved. Two other common fields in CDB are capability and security parameters.

When the execution of an OSD command results in errors, it returns a status of CHECK CONDITION, along with sense data to report errors generated in OSD logical units. The sense data contain information that allows initiators to identify the OSD object in which the reported error was detected. Any error can be reported by including the appropriate sense key and additional sense code to identify the condition. The OSD specification uses descriptor format sense data so that several sense data descriptors can be returned together.

IV. BASIC APPROACH

To capitalize on the parallelism in an active storage system, we designed and developed a Record Storage Application, the details of which are provided in the next section. The application was modeled to closely resemble a database system. The objective of this application development was to exemplify a framework that supports the parallel execution of any kind of application, and to analyze the performance gains of object oriented Active Disk computation over traditional single server systems.

A. Design Methodology

The basic design principles of developing an application to run in an Active Disk environment are to

- Exploit parallelism in the data-intensive portion of the data processing
- Separate the code to be executed at the disks as much as possible, from the code that controls the execution flow,so as to form self-contained and manageable units, and
- Take advantage of variations in available resources during execution, by using primitives.

These three principles allow for a great deal of performance and flexibility while placing and executing the application code. The Active Disk application code is divided such that the core piece of code will be made to run at each of the drives, while the high-level synchronization, control, or merging code is made to run at the host. This forms a client-server parallel programming model as illustrated in Figure 2. Initialization of input parameters takes place at the host and is sent across to all of the disks. Each of the disks operates on its own local data and produces its portion of the final result. The host collects the results from all the disks and merges them in an appropriate manner to form the final result. Since there is a portion of code that runs at the host and processes data from the disks, it is always possible to correct any incomplete or inaccurate computation at the drive, as long as the drive functions always act conservatively. When the drive is in doubt whether to include a particular record into its result set, it is expected to add the record anyway, so that the final filtering done at the host will be able to make a better judgment about it.

One of the major objectives of this work was to be able to facilitate the transfer of program files, related to an object type under consideration, to a OSD target which has no prior knowledge about that object type. For instance, the target could be ignorant of the methods that operate on a pdf or text file object. This capability allows any kind of function to be invoked by the client on any target. For this reason, we chose to implement our application in Java, thus allowing the methods pertaining to each object type to be encapsulated within its own class. In order for the OSD target to execute the methods, this Java code must first be downloaded to the target and then executed. This class file transfer is achieved

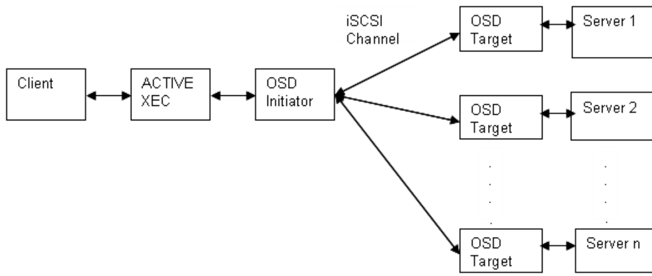

Fig. 2. Basic structure of Active Disk computation via OSD

by setting an attribute on the object where the attribute data is the JAR file associated with the class.

The class file, associated with the object as an attribute, defines the methods for that object. A client can execute methods on that object by using the OSD communication channel. The OSD T10 standard recently added the capability to specify the presence of a CDB continuation segment in the Data-Out buffer by including the CDB CONTINUATION LENGTH field to the basic OSD CDB format. This field specifies the number of bytes in the CDB continuation segment of the Data-Out buffer. The CDB CONTINUATION DESCRIPTOR TYPE field in the continuation segment specifies the format of the following CDB descriptor type specific data. A value of 0x0001h for the descriptor type supports scatter/gather features. The CDB continuation segment is followed by the command data segment, if any, in the Data-Out buffer. For our implementation we added a new CDB descriptor type, 0x0011h, to indicate that the continuation segment contains class file attribute data. The command data segment of the Data-Out buffer carries the method invocation associated with the command - i.e. the method name and parameters.

A java server runs on each OSD target, which takes in the requests coming from the client and first determines the type of object that the client request is meant for. Then the corresponding object's methods are executed. In the case of a class file transfer, the java server first downloads the transferred class file to an appropriate location and then uses the newly downloaded class to invoke the object's methods.

B. Sample APIs Developed

Two main functionalities that have been implemented in the application are the record sorting and matching record search functions, which are expected to benefit the most from such active computations. Sorting is a popular system benchmark and is most often done in the context of database systems, where it is usually combined with another operation.

We use at the client a variant of Merge Sort algorithm, which is the most common out-of-core sorting methodology. Merge Sort is based on the assumptions that: 1) a small list will take fewer steps to sort than a large list , and 2) fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. The algorithm performs a sequence of in-memory sorts on small subsets of the data, followed by a series of merge phases that combine the sorted subsets until the entire dataset is in order.

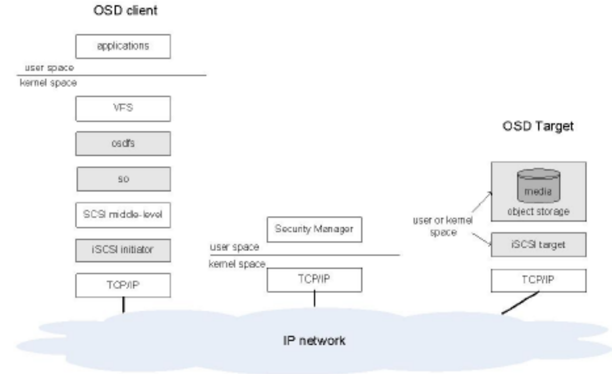

Fig. 3. Overview of reference implementation

C. Approach and Implementation

This section provides an outline of the structure of the application described above, that execute on Active Disks, including the design philosophy, the structure of the on-drive code and the host code. The data path of the commands is as shown in Figure 2.

Our implementation is based on the iSCSI OSD reference implementation developed by the University of Minnesota [12]. The reference implementation is built using Linux kernel modules and user-level libraries as shown in Figure 3. The kernel modules are used to implement a primitive OSD file system and the user-level code provides the communication channel and the implementation of the OSD target semantics. Since our framework does not require the use of the OSD file system, our implementation is entirely user based.

1) Client Design: The client side of our application was designed to have APIs for multiple operations on an object , that can exploit the parallelism offered by such a framework. Continuing with the above discussed examples of record sorting and searching, the splitting of the application code is done as follows. Each target sorts the data set it contains for the particular database object, in parallel, and transfers the sorted result back to the client. The client receives the sorted data sets from each of the targets and merges them together to produce the final sorted data set. This scheme of computation totally eliminates the need to transfer the data to the client for the sorting phase, as it is done directly at the target. In the case of record search, each target searches for records matching the given search criteria and sends back just the resulting records back to the client. The client, just as in the case of sorting, collects the partial result records from all the OSD targets and combines them together to produce the final search result. Since filtering of the data occurs at each of the targets, the data that gets transferred across the network is just a fraction of the initial data set, thereby reducing network

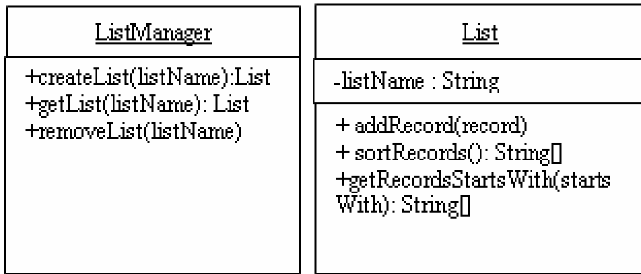

Fig. 4. Client side class diagrams

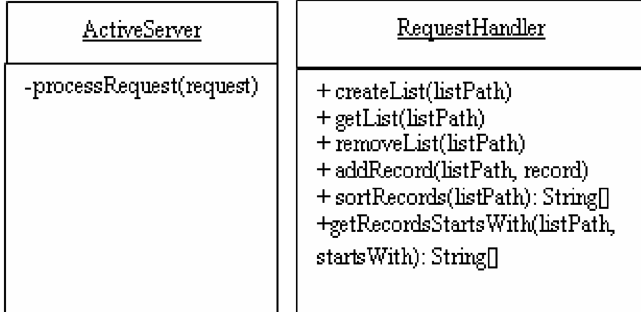

Fig. 5. OSD Target Server side class diagrams

bottleneck. The set of client side APIs developed for testing this framework for database objects is as shown in Figure 4. Similar client classes can be developed for any type of object, with APIs for the methods of interest.

2) OSD Target Server Design: The target server is designed to have a dedicated process, constantly listening to requests being sent from the client. In order to ensure the parallelism inherent with Active Disk applications, the client spawns a thread for each server, while sending requests. The individual server threads, send the requests to the client, and waits to receive the response from their respective targets. The main mentionable classes here, in the database object example, are the ActiveServer and the RequestHandler as shown in Figure 5. The ActiveServer class receives the client requests and calls the appropriate function in the RequestHandler which was downloaded from the client. The RequestHandler class has the methods for the object as described in the previous section.

In order to extend this to support multiple object types, a separate RequestHandler class has to be created for each object type. The appropriate RequestHandler class is then called by the ActiveServer class depending in the object type. This extensible architecture makes it very easy to add new objects and their functions to the framework. And as described above , the class file transfer capability allows the newly created RequestHandler classes of the objects to be transferred to the targets, along with the OSD commands. Thus extending the framework is made to become a trivial task.

The protocol used here for the client server communication is iSCSI. Configuration files have also been made use of, for specifying server parameters such as server names, port numbers, and directory paths for the object creation and so on.

3) OSD Design Changes: The OSD implementation used had to be tailored to fit the requirements of the application, in quite a few areas. The main idea adopted here was to relate the API of the object to an OSD command, such that issuing the OSD command from the OSD initiator to an OSD target, would be equivalent to sending a request from the client to a server via a TCP/IP socket connection. we chose to add a new OSD EXECUTE METHOD command for these APIs, in order to keep the method execution code self contained.

The interface between the Java application program client and the OSD client is implemented as data pipes. The same holds for the interface between the OSD target and the Java server program. When the application client issues a request, the ACTIVEXEC module receives the request and packages it into an appropriate EXECUTE METHOD command CDB. Application level parameters, if any, are placed as object attribute values in the CDB, and so is the class file data. Once the CDB reaches the OSD target, it is decapsulated and the parameters are passed to the java server, which takes up the execution of the requested method.

V. Results

The experimental setup we used to test out our application is a 16 node Linux cluster. One node was set up as the client machine and up to 8 nodes were used as OSD targets. Several test runs were conducted, with varying number of targets and varying data set sizes. Some of our experimental results are tabulated as below.

Table II show the results for a data set of 1 million records. The total sort time is the time required for the client to issue sort execute methods to the targets and then retrieve the data and do a merge sort locally. The average sort time is the average time required on each target node to do the read from disk and then sort. The results show that the targets sort times decrease linearly. The total sort time shows improvement but is not linear with respect to the number of target nodes. The reason for this is that the Java virtual machine begins to run against its heap size limit and as a result the merge time takes longer than expected.

We also measured the effect of using iSCSI as an RPC transport mechanism rather than a purely TCP/IP socket based implementation. We constructed a socket-based implementation that was hard coded to perform application-specific RPCs. Through experiments, we found that the iSCSI OSD implementation is comparable in run-time to the socket-based implementation with no more than 10% degradation in performance for the iSCSI implementation.

VI. Related Work

The basic idea of executing functions in processing elements directly attached to individual disks was explored extensively in the context of database machines such as CASSM [13], RAP [14], and numerous others [9]. These machines fell out of favor due to the limited performance of disks at the time and

No. of OSD Targets	Total Sort time (ms)	Avg. Target Sort time (ms)
1	19931	16689
2	12645	8716
4	8761	4227
8	7437	2351

TABLE II
RESULTS FOR 1M RECORDS

the complexity of building and programming special-purpose hardware that could only handle limited functions. Instead, database research has developed large-scale, shared-nothing database servers with commodity processing elements. It has recently been suggested that the logical extension is to perform all database processing inside programmable, "smart" system peripherals [15]. Recent commercial efforts such as products from DATAllegro and Netezza have revisited the notion of database/warehouse appliances and these products have shown significant promise.

Earlier work focused on database acceleration, and drawing from that work "active disks" emerged as a general parallel data processing framework. Work on active disks followed from previous work on network-attached secure disks (NASD), in which computational power at storage for parallel and network file system functions are exploited, as well as traditional storage optimizations [16], [17]. This work discussed several classes of applications that can benefit from Active Disks - including filters, multimedia, batching, and storage management - and enumerated the challenges to providing an execution environment on commodity disk drives [10], [18].

Acharya, et al. applied Active Disk ideas to a set of similar applications, including database select, external sort, data cubes, and image processing, using an extended-firmware model for next-generation SCSI disks [19]. This work was later expanded to large scale data processing using the concept of data filters in the DataCutter/Active Data Repository framework [20], [21]. Similarly, a group at Berkeley has independently estimated the benefit of Intelligent Disks for improving the performance of large SMP systems running scan, hash join, and sort operations in a database context [22].

Our work is most similar to recent work has examined the use of active disk principles in the Lustre parallel file system [23]. Their work uses a "filter" in the object storage target stack that processes streams of data from defined active objects. It differs from our work in that we allow the client to define an object-oriented class structure for an object and uses the OSD framework as an asynchronous RPC mechanism to access object methods. Thus, the execution is not dependent on operating on streams of data, but can operate on the entire object if necessary.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have demonstrated an object-oriented active storage framework built upon the iSCSI OSD standard. Many data-intensive applications have small CPU and memory requirements and are attractive for execution across Active Disks. Implementing this in the form of objects allows for better portability, reusability of components, extensibility and other such advantages of object oriented programming. A further improvement to this framework is to extend it to support any type of object. Currently the application supports only List objects. Another idea to pursue is to explore alternate methods for parallel programming, so as to reduce or even eliminate the overhead introduced by multi-threading, for small data sizes.

VIII. ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation High End Computing University Research Activity program under award number CCF-0621448.

REFERENCES

[1] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, no. 8, Aug. 2003.

[2] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, and D. Rochberg, "A Case for Network-Attached Secure Disks," Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, Tech. Rep. CMU-CS-96-142, Sep. 1996.

[3] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Communications of the ACM*, vol. 43, no. 11, pp. 37–45, Nov. 2000.

[4] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proceedings of the Annual Linux Showcase and Conference*, Oct. 2000, pp. 317–327.

[5] P. J. Braam and R. Zahir, "Lustre - A scalable high performance file system," Cluster File Systems, Inc., Mountain View, CA, Tech. Rep., July 2001.

[6] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu, "The Panasas ActiveScale storage cluster - delivering scalable high bandwidth storage," in *Proceedings of Supercomputing*, Pittsburgh, PA, November 2004, p. 53.

[7] *Information Technology - SCSI Object Based Storage Device Commands -2 (OSD-2)*, ANSI, Jan. 2008.

[8] H. Boral and D. DeWitt, "Database machines: An idea whose time has passed?" in *Proceedings of International Workshop on Database Machines*, Sep. 1983.

[9] D. DeWitt and P. Hawthorn, "A performance evaluation of database machine architectures," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Sep. 1981.

[10] E. Riedel, "Active disks - remote execution for network-attached storage," Ph.D. dissertation, Electrical and Computer Engineering, Carnegie Mellon University, 1999, tech. Report no. CMU-CS-99-177.

[11] *Information Technology - SCSI Architecture Model - 3 (SAM-3)*, ANSI, Sep. 2004.

[12] D. Du, D. He, C. Hong, J. Jeong, V. Kher, and Y. Kim, "Experiences in building an object-based storage system based on the OSD T-10 standard," Digital Technology Center, University of Minnesota, Minneapolis, MN, Tech. Rep. DTC 2006/13, 2006.

[13] S. Y. W. Yu and G. J. Lipvski, "CASSM: A cellular system for very large data bases," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1975.

[14] E. A. Ozharahan, S. A. Schuster, and K. C. Smith, "RAP: Associative processor for database management," in *Proceedings of AFIPS Conference*, 1975.

[15] J. Gray, "What happens when processing, storage, and bandwidth are free and infinite?" in *Proceedings of IOPADS*, Nov. 1997, keynote Address.

[16] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Jun. 1997.

[17] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Harding, E. Riedel, D. Rochberg, and J. Zelenka, "A cost-effective, high-bandwidth storage architecture," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[18] E. Riedel, G. A. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia," in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Aug. 1998.

[19] A. Acharya, M. Uysal, and J. Saltz, "Active disks: Programming model, algorithms and evaluation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[20] C. Chang, R. Ferreira, A. Sussman, and J. Saltz, "Infrastructure for building parallel database systems for multi-dimensional data," in *Proceedings of the International Parallel Processing Symposium*, 1999.

[21] M. Beynon, R. Ferreira, A. Sussman, and J. Saltz, "DataCutter: Middleware for filtering very large scientific datasets on archival storage systems," in *Proceedings of the IEEE/NASA Goddard Symposium on Mass Storage Systems and Technologies*, 2000.

[22] K. Keeton, D. A. Patterson, and J. M. Hellerstein, "The intelligent disk (IDISK): A revolutionary approach to database computing infrastructure," University of California at Berkeley, Tech. Rep., May 1998, white Paper.

[23] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in *Proceedings of Supercomputing*, Nov. 2007.