DOSAS: Mitigating the Resource Contention in Active Storage Systems

Chao Chen and Yong Chen
Department of Computer Science,
Texas Tech University,
Lubbock, Texas, USA
Email: chao.chen@ttu.edu, yong.chen@ttu.edu

Philip C. Roth
Computer Science and Mathematics Division,
Oak Ridge National Laboratory,
Oak Ridge, Tennessee, USA
Email: rothpc@ornl.gov

***Abstract*—Active storage provides an effective method to mitigate the I/O bottleneck problem of data intensive high performance computing applications. It can reduce the amount of data transferred as the application runs by moving appropriate computations close to the data. Prior research has achieved considerable progress in developing several active storage prototypes. However, existing studies have neglected the impact of resource contention when concurrent processes request IO operations from the same storage node simultaneously, which happens frequently in practice. In this paper, we analyze the impact of resource contention on active storage systems. Motivated by our analysis, we propose a novel Dynamic Operation Scheduling Active Storage architecture to address the resource contention issue. It offloads the active processing operations dynamically between storage nodes and compute nodes according to the system environment. By evaluating our architecture, we observed that: (1) resource contention is a critical problem for active storage systems, (2) the proposed dynamic operation scheduling method mitigates the problem, and (3) the new active storage architecture outperforms existing active storage systems.**

***Keywords*-dynamic active storage; active storage; resource contention; high performance computing; data intensive computing; parallel I/O; parallel file systems**

I. Introduction

Many high performance computing (HPC) applications from diverse areas such as astrophysics, climate modeling, medical image processing, and high-energy physics are becoming more highly data intensive than ever before. These applications transfer a large amount of data between the computer system's memory and storage. For example, the data volume processed in climate modeling, combustion, and astrophysics simulations can easily range from 100TBs to 10PBs [22]. Such large data volumes make input/output performance the key factor determining overall application performance in many modern high performance computing workloads.

Transferring such large data volumes between storage and compute nodes takes a considerable amount of time, even on today's highest-performing computer systems. Active storage was proposed to address this issue and has garnered intensive attention in recent years. Active storage avoids transferring large amounts of data by offloading data intensive computations to storage nodes and returning a relatively small result to compute nodes for further processing. It has been proven effective in reducing bandwidth requirements and improving the system performance in numerous prior studies [6, 13, 19, 22].

These existing studies, however, fall short when considering active storage for real-world applications in production environments. On a typical high end HPC system such as the Department of Energy's Jaguar system deployed at Oak Ridge National Laboratory (ORNL), the Intrepid system deployed at Argonne National Laboratory (ANL), and the National Science Foundation's Keeneland systems also deployed at ORNL, there may be dozens of applications running concurrently [9]. Future exascale systems are projected to have millions of nodes, with thousands of processing cores in each node. The total amount of concurrency of such exascale system designs is projected to be at least a billion threads of execution [2], significantly higher than current systems. Active storage systems for such high end HPC computing systems need to be able to process large numbers of I/O requests concurrently from a very large number of processes. Figure 1 illustrates that each storage node would receive a hugh amount of normal/active I/O requests from the processes of different applications. However, due to the fact that in such designs many compute nodes share a given storage node, the computation capability of a storage node is much smaller than that of a compute node. For example, in the ANL Intrepid system, 64 compute nodes share one I/O node [8]. In such a system, if a large amount of computation were offloaded, it would cause a serious resource contention at storage nodes with many compute nodes just waiting for results, which is completedly neglected by exsiting studies. Figure 2 illustrates the negative impact of this contention: compared to traditional storage, the performance of active storage is degraded when each storage node deals with more than 4 active I/O requests concurrently. Hence, we need to strike a balance between the computations offloaded to the storage nodes and the amount of time compute nodes have

* The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

to wait for results. This is essentially a scheduling problem, with the goal of splitting the computation part of active I/O requests between the storage nodes and compute nodes to minimize overall application run time for all applications in a workload.

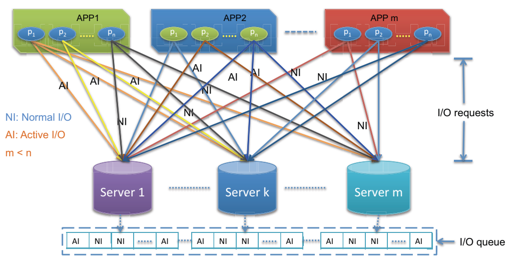

Figure 1. **Example showing contention caused by I/O requests from several applications**

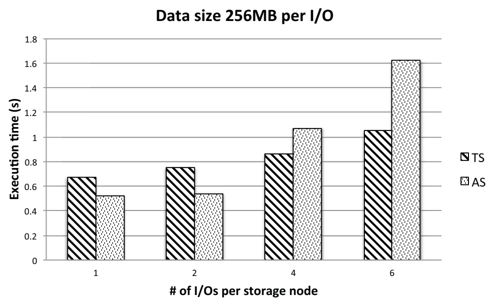

Figure 2. **Execution time of Gaussian filter under Traditional Storage (TS) and Active Storage (AS) model respectively, as the number of I/Os per storage node are increased. TS has better performance than AS in large scale I/O requests (TS: Traditional Storage, AS: Active Storage)**

In this study, we propose a novel *Dynamic Operation Scheduling Active Storage* (DOSAS) architecture to address the active storage resource contention problem. The proposed method offloads computations from compute nodes to storage nodes, but schedules these offloaded computations dynamically. The DOSAS system takes system status into account when offloading active I/O requests. When the storage node has available processing capability, it offloads and conducts computations for active I/O with its maximum capability. However when it is fully engaged with I/O services, normal I/O will take the priority and active I/O requests are changed to normal I/O requests by returning the computations to requesting compute nodes. In this manner, the new architecture reduces resource contention, increases system resource utilization, and improves overall system performance. Our evaluation of the proposed DOSAS architecture confirms the approach is effective in mitigating the contention problem.

The primary contributions of this paper are an investigation of the impact of resource contention within active storage systems (Section I), a proposed approach to mitigate the problem using a novel Dynamic Operation Scheduling Active Storage architecture (Section III), and an evaluation of the proposed architecture showing that it outperforms the existing active storage architectures (Section IV). We also briefly discuss related work in active storage, active disks, and parallel file systems (Section II) and some avenues for future research (Section V).

II. Related Work

Extensive studies have been done that focus on improving the performance of data-intensive HPC systems. These studies describe considerable successes in a number of forms.

Beginning in the late 1990s, Active Disk [1, 3, 4, 7, 10, 11, 17, 20, 23] was first proposed and explored in several studies to address the I/O bottleneck issue for data-intensive applications. As the name implies, its main idea was to use the processing power of the disk drive for computation. Both hardware architectures [3, 7, 10] and programming models [1, 17, 18] were proposed to address the problem. *Keeton et al* [10] introduced "Intelligent Disks" for decision support systems. it was equipped with a rudimentary communication component. *Lim et al* [11] proposed an Active Disk File System (*ADFS*) that offloaded operations from traditional central file servers to the disks. *Acharya et al* [1] proposed a stream-based programming model that allows application code to execute on the disks. However, these studies are designed to explore the power of embedded processors, and the approaches they explored have limited computation-offloading capability.

The idea of Active Storage is generated from Active Disk and has gained increasing attention in the past decade [6, 12, 13, 15, 16, 19, 21, 24–26]. It is proposed in the context of parallel file system environment to serve the same purpose with Active Disks. In 2005, *Felix et al*[6] presented the first real implementation of Active Storage on *Lustre* file system. Because it was implemented at the kernel level, it was not flexible, portable and readily deployable. *Piernas et al*[13] provided another implementation with a similar method purely in user space. In more recent work, they proposed a solution for the active storage to partially support the striped files. *Woo et al* [22] proposed a more complete implementation of the Active Storage on the *Parallel Virtual File System* (*PVFS*). Obviously, it has much more processing capabilities than Active Disks, and thus can offload more computations to reduce data movement.

These approaches are most relevant to our work. However, none of the existing studies considered the problem of resource contention when a large number of active I/O requests are offloaded to the same storage node. Such a problem is common in current HPC systems. The proposed dynamic operation scheduling and the active storage prototype built upon it aim to address this issue. The DOSAS system schedules the I/O requests on the fly to achieve better I/O system performance according to the current system status.

The MapReduce programming model and runtime system that move computation to data have proven to be very popular and have been shown to be effective for many data-intensive applications [5]. The MapReduce model, however, is typically layered on top of distributed file systems such as Google file system and Hadoop distributed file system. They are not designed for traditional high performance computing workloads. In contrast, our proposed active storage system inherits the features of active storage and is designed specially for high performance computing systems.

III. Design and Prototype of Dynamic Operation Scheduling Active Storage

Because the resource contention problem is not adequately addressed by existing active storage systems, we propose a novel Dynamic Operation Scheduling Active Storage (DOSAS) architecture. In this section, we first introduce the structure of the DOSAS and then describe its core components in detail. We also describe our prototype DOSAS system built using the PVFS2 [14] parallel file system.

A. DOSAS Architecture and Overview

In most high end HPC systems, there are separate storage nodes and compute nodes. This separation can provide better performance for parallel I/O. With the DOSAS architecture, we have assumed this system model.

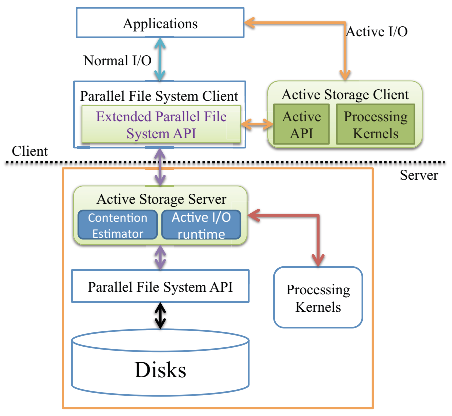

Figure 3. **the Overview of System Architecture**

As shown in Figure 3, the DOSAS architecture contains three major components: the *Active Storage Client (ASC)*, the *Active Storage Server (ASS)*, and *Processing Kernels (PKs)*. The *ASC* runs on the system's compute nodes as part of the application's I/O software stack. It provides an API for applications to request active I/O services, and also helps the storage nodes to complete the active I/O services when they are serviced as normal I/O requests. The *ASS* is placed on storage nodes, and is responsible for processing different I/O requests. The *PKs* are a collection of predefined analysis kernels that are widely used in data-intensive applications, and are deployed both at storage nodes and compute nodes.

For ease-of-use of the DOSAS system, we enhanced the MPI-IO interface. We choose MPI-IO for our prototype because MPI is widely used by scientific and engineering applications, and the MPI implementations found on most modern HPC systems include MPI-IO functionality. Choosing to base our interface on MPI-IO provides an easy migration path for those applications to effectively utilize the proposed approach.

In a DOSAS system, normal I/O is processed in the same way as it would be with a traditional parallel file system. In contrast, when an application issues an active I/O request using the enhanced MPI-IO interface, the request is transferred to the *Active I/O Runtime* (AIR) located on the system's storage nodes. The *AIR* is responsible for invoking the relevant processing kernels for further processing or for handling the active I/O request as a normal I/O request according to the system's status. A *Contention Estimator* (*CE*) periodically probes the system state, including CPU utilization, memory utilization and I/O queue. It is also in charge of generating the scheduling policy for active I/O requests and sending its decision, in the form of a scheduling policy, to the *AIR* component. The *AIR* then serves the I/O requests according to the scheduling policy it receives from the CE. If all the active I/O requests are rejected (perhaps because the storage node is already overloaded), the *AIR* will record and interrupt current active I/O being serviced (if there is one) and then return its computation to compute nodes and handle it as a traditional I/O request.

Our approach has two merits: First, it can dynamically offload the computations between compute nodes and storage nodes to achieve better performance of I/O system than a state-oblivious or static-scheduling approach. Second, it requires minimal changes to the application, thus not greatly increasing the complexity of application development to achieve the performance benefits of a DOSAS active storage system.

In the following sections, we detail architecture's three major components: the *Active Storage Client*, the *Active Storage Server* that is composed of *Contention Estimator* and *Active I/O Runtime*, and the *Processing Kernels*.

B. Active Storage Client

The *ASC* is a process that runs on the system's compute nodes. Its design is similar to the *pvfs2-client* of the *PVFS2* parallel file system, and it has two functionalities: serving as an interface for applications, and assisting the storage nodes to complete active I/O without the intervention of application developers when the I/O is treated as normal I/O by storage nodes.

Applications request active I/O operations from *ASC* with an enhanced *MPI-IO* interface. To support the DOSAS architecture, we have extended only one *MPI-IO* function. Our enhanced *MPI-IO* file call, *MPI_File_read_ex()*, is a simple extension to the existing *MPI_File_read()* call, and is similar to that introduced by Son *et al* [22]. The new API takes all the arguments in the original one and an additional argument that specifies the operations to be executed on the storage nodes. In addition, a simple structure type is used to encapsulate the *buf* arguments as shown in Table I. When the *ASC* receives an active I/O, it will register the operation, I/O size (from the *count* and the *datatype* arguments in the API) and its *fh* at local, and then transfer the request to the *AIR* through parallel file system *API*. When the *ASC* receives the result of the I/O, it will first check the $completed$ argument: if it equals 0, it will manage the rest of the processing until it has completed; if it equals 1, it will return the result to the requesting application process directly.

Table I
A COMPARISON OF THE DOSAS ENHANCED MPI-IO CALL WITH THE STANDARD CALL

```
struct result {
    bool completed; — 0: I/O not completed, 1 : completed
    void * buf;      — the saved result if completed or status of
                              operation if not completed
    MPI_File fh; — file handle (I/O uncompleted)
    long offset; — current data position
};

MPI_File_read_ex(MPI_File fh, struct result *buf, int count,
MPI_datatype, char *operation, MPI_Status * status);

MPI_File_read(MPI_File mpi_fh, void *buf, int count,
MPI_Datatype datatype, MPI_Status *status);
```

C. Active I/O Runtime

The *AIR* is a helper process that services the I/O requests according to the scheduling policy generated by the *CE*. It is in charge of two functions: invoking a relevant processing kernel when an active I/O arrives If the policy permits the active I/O requestes, and return the result or computation to the *ASC* once I/O operation is completed or interrupted respectively. To return back a completed I/O, *AIR* will set *completed* argument to 1 and fill the buf argument with result. To return back an uncompleted I/O operation, there are three cases to handle. For new arrival active I/O requests, *AIR* just set *completed* argument to 0 and let buf argument keep null. The request is now changed to be a normal I/O and will processed by *ASC*. For those requests still running at the system, The *AIR* should record the current status, including intermediate results, other useful variables of processing kernels before transferring them to the *ASC*. In order to simplify this procedure, the processing kernels is designed to record the status (the value of each variable) and send them back to *AIR* for filling the *buf* argument of *struct result*.

D. Contention Estimator and Scheduling Algorithm

One way to achieve the balance between the computation offloaded to the storage node and their computing capability is to schedule the active I/O requests dynamically taking the system's current workload into account. Inspired by this view, we use an algorithm that determines which active I/O requests can be offloaded. The *Contention Estimator* is an implementation of the algorithm. It monitors current system status, including I/O queue, memory usage and CPU usage, and generates the scheduling policy for all active I/O requests in current I/O queue by using the probed system information and the scheduling algorithm. It then sends its decision to *AIR* component for execution. In this subsection, we describe the proposed scheduling algorithm.

Table II
NOTATIONS AND DESCRIPTIONS

n	the number of I/O requests in I/O queue
k	the number active I/O requests in I/O queue
d_i	the request data size of i-th I/O request
D_A	the total data size requested by active I/O requests. Thus $D_A = \sum_i d_i$ if(ith I/O is active I/O)
D_N	the total data size requested by normal *I/O* requests. Thus $D_N = \sum_i d_i$ if(i-th I/O is Normal I/O)
D	the total request data size in I/O queue. Thus $D = \sum_{i=1}^{n} d_i$ and $D = D_A + D_B$
$S_{C,op}$	the computation capability of each storage node given operation op
$C_{C,op}$	the computation capability of each compute node given operation op
$f(x)$	the time need to compute on x size data
$g(x)$	the time needed to transfer x size data from storage node to compute node
$h(x)$	the data size of the result computed on x size data by active I/O
bw	the bandwidth of compute-storage network

1) Assumptions and Definitions: When designing the DOSAS scheduling algorithm, we made several assumptions to control the algorithm's complexity. These assumptions, though providing a simplified abstraction of I/O behavior, are reasonable in practice.

Assumptions:

- Each process requests one I/O operation at a time.
- Each I/O can be identified with its request data size and I/O type (active I/O or normal I/O).
- The workload of an application consists of two separate parts: computation (time spent for data processing) and data movement (time spent for transferring the data from storage nodes to compute nodes).

Table II introduces the notations used in our scheduling algorithm. The value of $S_{(C,op)}$ used in the algorithm is estimated by the *CE* according to its max value (achieved when a storage node is fully dedicated to executing the *op*) and the current system environment.

Based on our assumptions listed earlier, we can estimate the total execution time of the current I/O requests. For example, if all active I/O requests are carried out, the total execution time would consist of the time of computation of active I/O requests, the time transferring the results of active I/O requests, as well as network transfer time of data for normal I/O requests. Symbolically, this time T_A is:

$$T_A = f(D_A) + g(D_N) + g(h(D_A)); \qquad (1)$$

If all I/O requests are processed as normal I/O, all of the computation is transferred to and parallelly executed on compute nodes, the execution time would be expressed as:

$$IO\_size = max(d_i) \qquad (i - th\ I/O\ is\ active\ I/O) \quad (2)$$

$$T_N = g(D) + f(IO\_size); \quad (3)$$

Both $f(x)$ and $g(x)$ are linear functions, and $f(x) = \frac{x}{S_{C,op}}$ for each storage node and $f(x) = \frac{x}{C_{C,op}}$ for each compute node. Meanwhile, $g(x)$ is defined as a function of data size and network bandwidth: $g(x) = \frac{x}{bw}$.

2) The I/O Scheduling Algorithm in DOSAS: Using the assumptions and definitions above, we model the scheduling problem as a binary optimization problem. For every active I/O request, a storage node has two choices: doing active I/O as requested or doing normal I/O. Therefore, to serially process each I/O queue of k active I/O requests, there are 2^k combinations. The optimization goal is to minimize the total time to finish all requested active I/O requests:

$$t = \sum_{i=1}^{k} [x_i a_i + y_i (1 - a_i)] + z \qquad (4)$$

Where,

$$x_i = \frac{d_i}{S_{(C,op)}} + \frac{h(d_i)}{bw} \qquad (5)$$

$$y_i = \frac{d_i}{bw} \qquad (6)$$

$$z = \frac{\max \quad d_i (1 - a_i)}{C_{(C,op)}} \quad \forall i \in [1..k] \qquad (7)$$

and, $a_i = 1$ or $a_i = 0$ if the i-th requested active I/O is done as requested and as normal I/O, respectively. Hence, the optimization problem is:

$$\begin{aligned} \underset{[a_1,\ldots,a_k]}{\text{minimize}} \quad & \sum_{i=1}^{k} [x_i a_i + y_i(1 - a_i)] + z \\ \text{subject to} \quad & a_i \in \{0, 1\}, \forall i \in [1..k]. \end{aligned} \qquad (8)$$

This problem can be solved using general constraint programming solver. An alternative but naïve solution is to try all possible combinations and pick the one that minimizes the target function. This can be done as follows: let $X = [x_1, \ldots, x_k]$, $Y = [y_1, \ldots, y_k]$ and $Z = [d_1, \ldots, d_k]$, we create a matrix to store all possible permutations to process active I/O requests:

$$A = \begin{bmatrix} a_{11} & \ldots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{k1} & \ldots & a_{km} \end{bmatrix} \qquad (9)$$

where $a_{ij} \in \{0, 1\}, \forall i \in [1..k]$ and $\forall j \in [1..m], m = 2^k$. $a_{ij} = 1$ and $a_{ij=0}$ if the i-th requested active I/O is done as requested and as normal I/O, respectively, in the j-th combination.

Hence, each column of A, denoted as A_j, represents a unique combination for handling requested active I/O requests. In this sense, we require any two columns of A to be different, thus $A_j \neq A_p$ if $j \neq p$.

The values of target function (Eq. 8) for different combinations can be computed by

$$X \cdot A + Y \cdot B + \frac{\max X \cdot B}{C_{C,op}} \qquad (10)$$

which is a 1-by-m vector. B is a matrix obtained from A such that $b_{ij} = 1 - a_{ij}, \forall i \in [1..k]$ and $\forall j \in [1..m]$.

The optimization combination is then:

$$\arg\min_{j} \quad X \cdot A + Y \cdot B + \frac{\max X \cdot B}{C_{C,op}} \qquad (11)$$

E. Processing Kernels

The *Processing Kernels* component in the architecture is a collection of predefined analysis kernels that are widely used in data-intensive applications. We adopt a similar design to the work of Son *et al* [22], as it is simple and easy to implement. Unlike other active storage architectures, our approach employs a *PKs* component both at the client side and storage side. Using this strategy, the system can process active I/O at storage nodes or compute nodes transparently to applications. When the storage side is fully loaded, the client side processes the active I/O requests automatically without further application intervention, simplifying applications development. For performance reasons, the *PKs* component in our design communicates with the *AIR* through shared memory, a technique widely used for inter-process communication within a given compute node. Although our approach avoids increasing the complexity of application development, it unavoidably increases the complexity of the Processing Kernels implementation. Fortunately, each kernel just needs to be implemented once and can be used many times. When a kernel receives a terminating signal from the *AIR*, it will write the shared memory with its status, including the values of all variables in the form $variable\_name, variable\_type, value$, and then send a signal indicating the kernel's termination to the *AIR*.

IV. Evaluations and Result Analysis

We evaluated our DOSAS ideas by implementing a prototype DOSAS system, and then observing its performance using several benchmarks. In this section, we detail that evaluation and explain our results.

Table III
BENCHMARKS

Benchmark	Computation Complexity	Processing Rate
SUM	1 addition operation per data item	860 MB/s
2D Gaussian Filter	9 multiplication operations, 9 addition operations and 1 divide operation per data item	80 MB/s

A. Experiment Platform

1) Hardware Platform: To demonstrate the potential benefits of our prototype, we have performed experimental evaluations on the *Discfarm cluster* at the Texas Tech University. The *Discfarm cluster* is composed of one Dell PowerEdge R515 rack server node and 15 Dell PowerEdge R415 nodes, with a total of 32 processors and 128 cores. The nodes are fully connected using 1 Gigabit Ethernet and the tested network bandwidth in experiments is 118MB/s. We only used PowerEdge R415 nodes, thus the storage node and the compute node have the same processing capability in our evaluations.

In our experiments, we simulated each storage node with 2 cores and each process requesting I/O operations with the same data size. As described in Section III, the final decision of *DOSAS* is mainly related to active I/O requests. Therefore, we evaluated the situations when each storage node processes 1, 2, 4, 8, 16, 32 and 64 active I/O requests, and each I/O requesting 128MB, 256MB, 512MB and 1GB data respectively. We take the total execution time of all I/O requests, but not the single I/O, to verify the system performance. In order to simplify the experiments, we used one benchmark but ran it with multiple instances each time. Therefore, we only need to generate at most 64 combinations. Although this approach simplifies the experiments, it does not affect the experiment results that illustrate the impact of resource contention for active storage, and that verify the effectiveness of the proposed DOSAS architecture.

2) Evaluated Benchmarks: We evaluated two benchmarks in our experiments. They have completely different computation complexity as described in Table III. The *SUM* benchmark has little computation workload and just needs 1 addition operation for each data element. In contrast, the *2D Gaussian Filter*, which is wildly used in the area of geographic information systems and medical image processing, has much more computation workload than the *SUM* benchmark. It needs 9 multiplication operations and 9 addition operations for each data element. We tested the processing capability of a core for each benchmark, and found that each core could process 860MB data per second for the *SUM* benchmark and 80MB data per second for the *2D Gaussian Filter*.

3) Evaluated Analysis Schemes: To demonstrate the effectiveness of our proposed active storage system, we tested three schemes:

- Traditional Storage (TS): In this scheme, the servers are responsible for normal I/O operations. The analysis kernels are executed at the clients.
- Normal Active Storage (AS): In this scheme, the kernels are always executed at server side, with each storage node process requested kernels.
- Dynamic Operation Scheduling Active Storage (DOSAS): In this scheme, the I/O operations are dynamically scheduled according to the system situation of storage nodes.

B. Experiment Results

We first show the negative impact of resource contention on the AS scheme and then we present the evaluation of the dynamic operation scheduling algorithm. We then analyze of our proposed scheme by comparing overall execution time with two other schemes and the bandwidth comparison of each scheme.

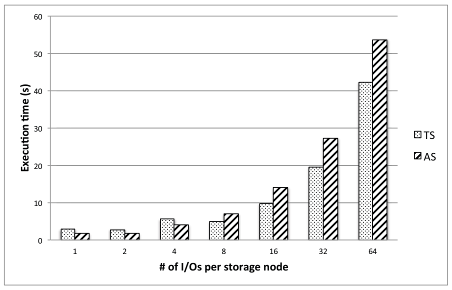

Figure 4. Execution time of 2D Gaussian Filter under AS and TS scheme with increasing I/O requests, each I/O requests 128MB data

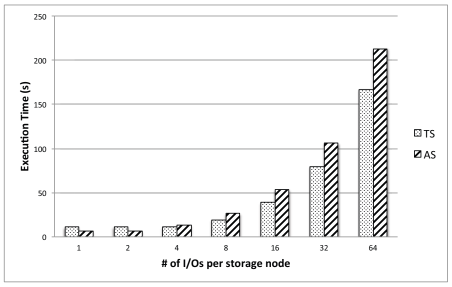

Figure 5. Execution time of 2D Gaussian Filter under AS and TS scheme with increasing I/O requests, each *I/O* requests 512MB data

1) Performance Impact of the Resource Contention on AS: There is a negative performance impact from resource contention with active storage systems. Figure 4 and Figure 5 show the execution time of the *2D Gaussian Filter* benchmark running with AS and TS respectively. These results show that, when there is a few I/O requests, AS has better performance (lower execution time) than TS, because each storage node has sufficient capability for processing, and reducing the data movement. This is the primary reason that AS achieved better performance than TS in such situation. However, when more I/O requests competed for limited resources, TS achieved better performance. The reason for the performance degradation of AS is attributed to the increasing amount of computation of active I/O requests overloads the

Table IV
SCHEDULING ALGORITHM EVALUATION

Situations	Algorithm Decision	Practice	Judgment
1	Active	Active	TRUE
2	Active	Active	TRUE
3	Active	Normal	FALSE
4	Normal	Normal	TRUE
5	Normal	Normal	TRUE
6	Normal	Normal	TRUE
7	Normal	Normal	TRUE
8	Active	Active	TRUE
9	Active	Active	TRUE
10	Active	Normal	FALSE
11	Normal	Normal	TRUE
12	Normal	Normal	TRUE
13	Normal	Normal	TRUE
14	Normal	Normal	TRUE
15	Active	Active	TRUE
16	Active	Active	TRUE
17	Active	Active	TRUE
18	Active	Active	TRUE
19	Active	Active	TRUE

storage nodes, while the compute nodes wait for the results. This idle waiting wastes computation resources of compute nodes. These results show a clear need for a system that can dynamically schedule the I/O operations as normal I/O or active I/O, fully using the resources of storage nodes without overloading them.

We have also evaluated the impact of computation complexity on the performance of the AS scheme, and found that when the computation complexity of a kernel is low enough, AS can always achieve better performance than TS for all scale sizes. For example, as shown in Figure 6, AS scheme always achieved better performance under all tested I/O scale size. This was because the *SUM* benchmark has very low computation complexity, and each core can process as may as 860MB data per second, which is much larger than the network bandwidth (118MB/s). Thus, *AS* can reduce considerable amount of time than *TS* in such situation.

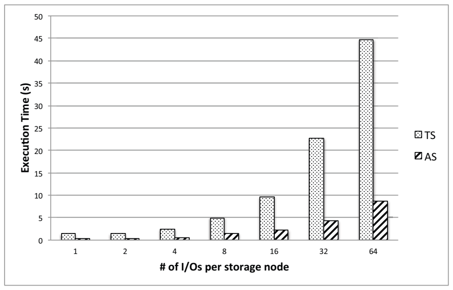

Figure 6. Execution time of SUM benchmark under AS and TS schemes with increasing I/O requests, each I/O requests 128MB data

2) Scheduling Algorithm Evaluation: We have evaluated the scheduling algorithm of DOSAS with *SUM* and *2D Gaussian Filter* benchmarks. With each benchmark requesting different numbers of I/O requests and each I/O requesting different data sizes, we generated 64 situations to evaluate the algorithm. The algorithm outputs correct decisions in 95% of the situations. Table IV shows some of the situations we considered; we are unable to show all due to space limitations. It is worth noting that our algorithm achieved 100% accuracy for the *SUM* benchmark. However, it misjudged the *2D Gaussian Filter* at the boundary where I/O scale slides from small to large (4 processes per storage node in our experiments). The reasons for this misjudgment are twofold: (1) system variation: for example, the network bandwidth is not always fixed in practice and ranged from 111MB/s to 120MB/s depending on the system and network environment; (2) the algorithm itself: the algorithm has a limitation that it only considers the processing and network transfer time. Other factors, such as the system task scheduling and network latency, are not considered. Although the algorithm was not perfect, we feel it has provided a reasonable estimation (with 95% accuracy) and decision based on considered factors without too much complexity.

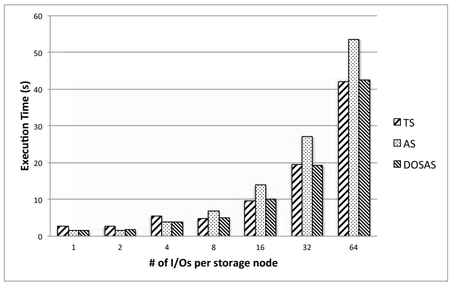

Figure 7. Performance of DOSAS and its comparison with AS and TS (each I/O requests 128MB data)

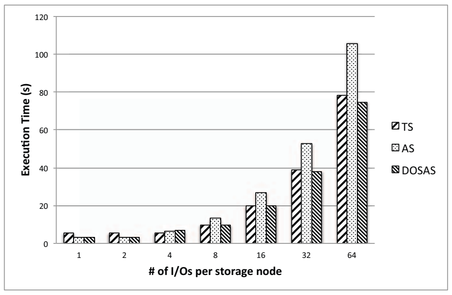

Figure 8. Performance of DOSAS and its comparison with AS and TS (each I/O requests 256MB data)

3) Performance Analysis of DOSAS: Figures 7 to 10 compare the execution time of our DOSAS prototype with the AS and TS approaches for a range of sizes. The I/O request of these four sets of experiments was 128MB, 256MB, 512MB, and 1GB data respectively. From the figures, we can see that the *AS* achieved superior performance for small scale size, whereas the *TS* outperformed the AS at large scale size. Our approach combines the merits of both and improved the overall system performance at both small and large scale sizes benefiting from the embedded dynamic operation scheduling. If not considering the measurement variations, the DOSAS achieved roughly the same performance with the AS scheme when there was little resource contention, and gained about 40% performance improvement compared to the TS scheme. Meanwhile, the DOSAS achieved nearly equal performance to the TS scheme when there were more I/O requests, and gained about 21% performance improvement compared to the AS scheme. This performance gain was attributed to the dynamic I/O scheduling that can schedule I/O requests as either active I/O or normal I/O according to the current system status.

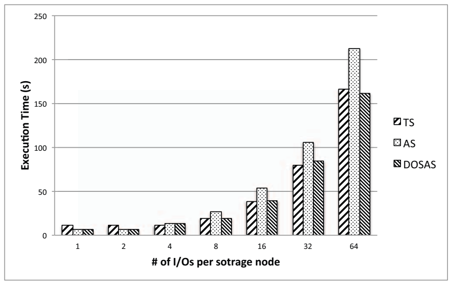

Figure 9. Performance of DOSAS and its comparison with AS and TS (each I/O requests 512MB data)

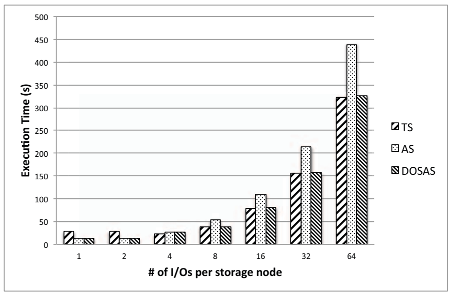

Figure 10. Performance of DOSAS and its comparison with AS and TS (each I/O requests 1GB data)

4) Bandwidth: We also compared the bandwidth achieved with DOSAS against that achieved with both TS and AS. These experiments were also conducted using the *2D Gaussian Filter* Benchmark with each I/O requests 256MB and 512MB data respectively. The number of I/O requests per storage node also ranged from 1 to 64. Figure 11 and Figure 12 show that the AS scheme has a better bandwidth than the TS for small I/O scale sizes, but vice versa for large I/O scale sizes. The DOSAS was able to identify the contention and handle it properly, thereby achieving the best performance with nearly all I/O scale sizes.

In summary, our evaluation results show the DOSAS approach outperforms the two existing schemes AS and TS. It presents a novel solution for achieving high I/O performance and shows great potential for alleviating I/O resource contention problems that are expected to only grow worse with increases in HPC system scale.

V. Conclusion and Future Work

Active storage provides a promising solution to mitigate the I/O performance problem for data-intensive high-end/high-performance computing applications. It can reduce the bandwidth cost by moving appropriate computations from compute nodes to storage nodes. Prior research, however, ignored the importance of resource contention when a number of active I/O requests are offloaded, which could cause overloading of storage nodes and subsequent degradation of overall system performance.

This paper proposes a *Dynamic Operation Scheduling Active Storage (DOSAS)* architecture that considers resource contention in active storage systems. We have demonstrated and verified that resource contention in active storage servers has a clear impact on the system performance. We have presented a new DOSAS system to address this issue. The DOSAS has a contention estimator component embedded and makes the offloading decision dynamically on the fly according to the system situation. Our results show that the proposed DOSAS scheme outperformed existing active storage architectures, and could serve as part of a high performance I/O subsystem for current and future high-end HPC systems.

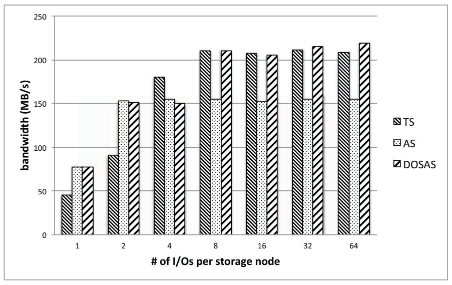

Figure 11. Bandwidth achieved of each scheme with each I/O requesting 256MB data

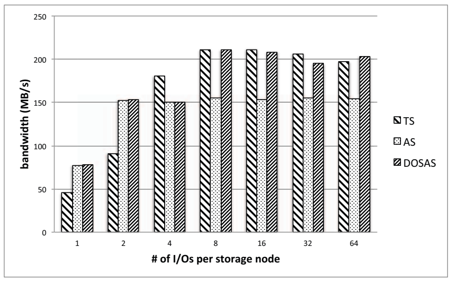

Figure 12. Bandwidth achieved of each scheme with each I/O requesting 512MB data

VI. Acknowledgment

This research is sponsored in part by the Oak Ridge Associated Universities (ORAU) Ralph E. Powe Junior Faculty Enhancement Award and by the National Science Foundation under grant CNS-1162488. The authors acknowledge Forrest Sheng Bao for his contribution to the scheduling algorithm, and the High Performance Computing Center (HPCC) at Texas Tech University at Lubbock for providing HPC resources that have contributed to the research results reported within this paper. URL: http://www.hpcc.ttu.edu.

References

[1] Anurag Acharya and Joel Saltz. Active Disks : Programming Model , Algorithms and Evaluation. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998.

[2] Yong Chen. Towards scalable i/o architecture for exascale systems. In *4th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*, 2011.

[3] Steve Chiu, Wei-keng Liao, and Alok Choudhary. Design and Evaluation of Distributed Smart Disk Architecture for I/O-Intensive Workloads. In *Proceedings of International Conference on Computational Science*, 2003.

[4] Gregory Chockler and Dahlia Malkhi. Active Disk Paxos with infinitely many processes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[6] Evan J Felix, Kevin Fox, Kevin Regimbal, and Jarek Nieplocha. Active Storage Processing in a Parallel File System. In *6th LCI International Conference on Linux Clusters: The HPC Revolution*, Chapel Hill, North Carolina, 2005.

[7] Mark Franklin, Roger Chamberlain, Michael Henrichs, Berkley Shands, and Jason White. An Architecture for Fast Processing of Large Unstructured Data Sets. In *Proceedings of the IEEE International Conference on Computer Design*, 2004.

[8] http://www.alcf.anl.gov/intrepid.

[9] http://status.alcf.anl.gov/intrepid/activity.

[10] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, 1998.

[11] Hyeran Lim, Vikram Kapoor, Chirag Wighe, and David Du, H.-C. Active Disk File System: a Distributed, Scalable File System. In *18th IEEE Symposium on Mass Storage Systems and Technologies (MSS '01.)*, pages 101–116, San Diego, CA, USA, 2001. IEEE Computer Society.

[12] Juan Piernas and Jarek Nieplocha. Efficient Management of Complex Striped Files in Active Storage. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, 2008.

[13] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[14] http://www.pvfs.org/.

[15] Lingjun Qin and Dan Feng. Active Storage Framework for Object-based Storage Device. In *20th International Conference on Advanced Information Networking and Applications*, pages 97–101, 2006.

[16] Brandon Rich and Douglas Thain. DataLab: Transactional Data-Parallel Computing on an Active Storage Cloud*. In *IEEE/ACM High Performance Distributed Computing*, 2008.

[17] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *In Proceedings of the 24rd International Conference on Very Large Data Bases(VLDB '98)*, New York, NY, USA, 1998.

[18] Erik Riedel, Garth A. Gibson, and David Nagle. Active Disks for Large-Scale Data Processing. *IEEE Computer*, 2001.

[19] Muthian Sivathanu, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002.

[20] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST*, 2003.

[21] Clinton Wills Smullen, Shahrukh Rohinton, Sudhanva Gurumurthi, Parthasarathy Ranganathan, and Mustafa Uysal. Active Storage Revisited : The Case for Power and Performance Benefits for Unstructured Data Processing Applications. *Proceedings of the 5th conference on Computing frontiers*, 2008.

[22] Seung Woo Son, Samuel Lang, Philip Carns, Robert Ross, and Rajeev Thakur. Enabling Active Storage on Parallel I / O Software Stacks. In *26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[23] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Evaluation of Active Disks for Decision Support Databases. In *Proceedings of the 6th International Symposium on High-performance Computer Architecture*, pages 337–348, 2000.

[24] Rajiv Wickremesinghe, Jeffrey S Chase, and Jeffrey S Vitter. Distributed Computing with Load-Managed Active Storage. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing(HPDC-11)*, 2002.

[25] Yulai Xie, Dan Feng, and Darrell D E Long. Design and Evaluation of Oasis : An Active Storage Framework based on TIO OSD Standard. In *27th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2011.

[26] Yu Zhang and Dan Feng. An Active Storage System for High Performance Computing. *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, pages 644–651, 2008.