Active Burst-Buffer: In-Transit Processing Integrated into Hierarchical Storage

Chao Chen
Georgia Institute of Technology
Email: chao.chen@gatech.edu

Michael Lang
Los Alamos National Lab
Email: mlang@lanl.gov

Latchesar Ionkov
Los Alamos National Lab
Email: lionkov@lanl.gov

Yong Chen
Texas Tech University
Email: yong.chen@ttu.edu

***Abstract*—The data volume of many scientific applications has substantially increased in the past decade and continues to increase due to the rising needs of high-resolution and fine-granularity scientific discovery. The data movement between storage and compute nodes has become a critical performance factor and has attracted intense research and development attention in recent years. In this paper, we propose a novel solution, named *Active burst-buffer*, to reduce the unnecessary data movement and to speed up scientific workflow. Active burst-buffer enhances the existing burst-buffer concept with analysis capabilities by reconstructing the cached data to a logic file and providing a MapReduce-like computing framework for programming and executing the analysis codes. An extensive set of experiments were conducted to evaluate the performance of Active burst-buffer by comparing it against existing mainstream schemes, and more than 30% improvements were observed. The evaluations confirm that Active burst-buffer is capable of enabling efficient data analysis in-transit on burst-buffer nodes and is a promising solution to scientific discoveries with large-scale data sets.**

I. Introduction

Modern scientific discoveries normally consist of two phases including simulation and analysis. They rely on high-performance computing (HPC) facilities to run simulation codes, and outputs will be stored and read later for analysis to mine scientific insights. In existing frameworks, data analysis tasks are normally taken offline on a small cluster after the completion of simulations, which requires a data movement from storage nodes to analysis nodes.

With the increasing demands of high resolution, fine granularity, and multi-model probabilistic scientific discoveries, scientific applications are becoming more and more data-intensive. The HPC systems that support these applications have increasingly larger memory footprints currently exceeding a petabyte and multiple petabytes in the near future. This results in an explosion of scientific data that has to be stored and analyzed with different methods to extract scientific insights. Examples of such scientific simulations include climate sciences [1], fusion physics [2], and astrophysics [3]. In a single simulation run, each of them produces hundreds of terabytes of data currently and more in the near future.

Current HPC systems, however, can not afford enough I/O bandwidth for effectively writing/reading such a huge amount of data in a desired amount of time required by modern scientific discoveries. I/O bottleneck has been identified as one of the main performance bottlenecks and energy inefficiency issues. In the upcoming Exascale system, this issue will become even more important because of the increasing performance gap between the computing and I/O, and has attracted intensive research in recent years [4–11].

Burst-buffer [8] is an exciting solution for improving I/O performance. Observing that the I/O patterns of scientific applications are not uniform but are bursty, burst-buffer is designed as a hierarchical storage which adds a high speed storage caching tier, such as Solid State Drives (SSDs), to allow the application to write data quickly, so it can return to computation as soon as possible. The data can then be asynchronously transferred from the burst-buffer to the permanent storage system without interrupting the application. This movement can be done on a separate network, enabling a high degree of overlap between the computation and I/O phases of applications. Burst-buffer can dramatically reduce the application-aware I/O time and improve the simulation performance. However, the need to retrieve a huge volume of data to analysis nodes is still inefficient in both performance and power for scientific discoveries.

In this paper, we propose an *Active burst-buffer* idea to reduce system wide data movement and speed up scientific discovery. Following the Exascale I/O framework proposed in ongoing FastForward project [5], Active burst-buffer is designed as a further enhancement to existing burst-buffer concept [8]. Observing that burst-buffer (e.g. SSDs) has a large capacity and can cache more data for a relative long time, Active burst-buffer enables incremental analysis on the data cached in burst-buffer. It manages the cached data in burst-buffer as a logical file for analysis codes, and provides a MapReduce-like computation framework to analyze the cached data before they are finally written to the storage system. Compared to the normal paradigms of scientific discoveries that analyze the data offline after simulations, Active burst-buffer can considerably reduce data movement between the storage system and the computing system, saving I/O bandwidth and speeding up the scientific discoveries. The current experimental results of Active burst-buffer are promising. With small overheads for single I/O operation, it can reduce data movement and speed up the scientific analysis by around 30%.

Contributions: To the best of our knowledge, this work is the first study on enabling analysis in burst-buffer nodes. The contributions of this work are as follows: 1) we present the design of Active burst-buffer, a method to explore the compute power in burst-buffer nodes for in-transit analysis; 2) we built

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

an Active burst-buffer prototype based on the FUSE library, and evaluated its performance with a set of experiments; 3) via prototyping and evaluation, we show the performance of Active burst-buffer is effective in reducing data movement and enabling scientific discoveries across large scientific data-sets.

The rest of the paper is organized as follows: Section II introduces more detailed background and the motivation of this work. Section III presents the design of Active burst-buffer. Section IV presents the prototyping and the evaluation of Active burst-buffer. Section V discusses the existing related work, and Section VI concludes this study.

II. Background and Motivation

In this section, we introduce the background and motivation of designing Active burst-buffer.

A. Exascale I/O Challenges and Burst-buffer

The storage capacity of the coming Exascale HPC system is projected to be more than 960PB [10], which poses significant challenges in the design of I/O and storage subsystems. To meet the requirements of applications, and match the computing performance (FLOPs), the projected I/O bandwidth demand is around 106 TB/s, which is about 4000x more than existing peak bandwidth (0.027 TB/s) [10]. If we continue to use mechanical disks, the number of disks required for such high bandwidth will cost about $200M dollars [9, 10]. Similarly, if we use Solid State Drives (SSDs) for bandwidth, the number of SSDs required for such large capacity will cost even more. Therefore, a hybrid storage system, that uses SSDs for bandwidth and Hard Disk Drives for capacity, is considered to be the only practical answer ($50M) [10].

The burst-buffer is an exciting design for the Exascale I/O system. It has been well studied that HPC applications have bursty I/O patterns [12, 13]. Typically, HPC applications issue and complete the I/O requests in a busy, short time, and then conduct the computing phase for a long time. For 98% of the time, the I/O system was utilized at less than 33% of peak I/O bandwidth [13] . While applications issue I/O in a bursty patterns, the burst-buffer will quickly absorb I/O requests in local SSDs, and let the applications return to computing as soon as possible. The burst-buffer will then write the data stored in local SSDs to a parallel file system as the applications continue to the next cycle of computing. It overlaps the computing phase and actual I/O phase of applications and reduces the application aware I/O time.

Considering that burst-buffer could have 100s GB storage capacity, and the data can be cached for relative long periods, we propose an Active burst-buffer to conduct the analysis on the cached data while the data is resident on the burst-buffer. Active burst-buffer is designed as an enhancement to Exascale I/O software stack. It shares a similar design with other in-situ/in-transit analysis approaches [14]. But it provides more powerful analysis through the MapReduce-like analysis framework, and our design allows full utilization of the available system resources without introducing dedicated nodes.

B. Motivations

The design of Active burst-buffer is directly motivated by a coarse-fine execution paradigm widely used in existing scientific discoveries. Due to the performance limitation of existing HPC systems, scientists currently prefer the coarse-fine paradigm that runs the simulation several times with different configurations to get the desired results. They first run the simulation with a coarse resolution for a large-scale area, and then analyze the data to find a small interest area, and rerun the simulation with a fine resolution upon the small interest area. They repeat these phases until they get the desired results. This paradigm will be possibly extended to Exascale computing due to at least two reasons. Firstly, in upcoming Exascale systems, the I/O subsystem has been recognized as one of the main bottlenecks. The computing resource will be relatively cheap, and the data movement will be expensive. This paradigm is a good way to reduce the volume size of outputs and avoid unnecessary data movements. Secondly, burst-buffer improves application aware I/O performance mainly through overlapping computing and I/O phases. However, in Exascale systems, the concurrency of each node will be improved around 800x more than existing leading HPC systems, while the memory capacity is projected to be improved by only 200x. Therefore, the memory per core in each node will shrink considerably, which will cause much more frequent I/O requests. The total data size, however, is not decreasing but increasing significantly due to the scale of nodes. These changes will ultimately and undoubtedly limit the performance achieved by burst-buffer. The coarse-fine paradigm can effectively reduce the workload of burst-buffer and improve the system performance.

In conclusion, the coarse-fine paradigm can avoid large amounts of unnecessary data movements by avoiding producing uninterested area data. However, each copy of generated data needs to be accessed at least twice happened at simulation phase and analysis phase respectively. Considering the application scale in Exascale systems, the generated overhead could be very high, which not only impacts the application itself, but also the entire system.

Active burst-buffer is introduced to speed up this process. With Active burst-buffer, users can analyze data step-by-step on the I/O path without waiting for the completion of simulation runs. It considerably reduces the data movement from the storage system to analysis clusters and conserves the I/O bandwidth for other applications.

III. Design of the Active burst-buffer

A. Overview

Active burst-buffer is a virtual file system accompanied by an analysis framework. It resides on each burst-buffer node (or I/O forwarding nodes in Blue Gene systems) to manage the burst-buffers' storage, as illustrated in Figure 1. Inherited from the I/O forwarding architecture, each burst-buffer node serves a set of compute nodes, and each compute node will forward its I/O operations to the designated burst-buffer node

through the I/O forwarding middleware (e.g., IOFSL [4]). There is no overwriting on the data cached in burst-buffer. The Active burst-buffer operates between the I/O forwarding service and the parallel file system client. At each bursty I/O request cycle, the I/O forwarding service will receive the data from the simulation processes and cache it to its local SSD, through the Active burst-buffer. Then, Active burst-buffer will move the data to the parallel file system while applications continue to the next phase of computation. Before the data is actually written to the file system, the Active burst-buffer can invoke a user-implemented analysis code, if one is provided, to analyze the cached data, and save intermediate results in a designated file. When the simulation is complete, the Active burst-buffer can invoke another user-implemented analysis code to aggregate the intermediate results and get the final result. Active burst-buffer is fundamentally different from traditional cache systems in the sense that: 1) it acts like an "advanced" or "active" write-back cache with the integration of in-transit analysis; 2) there are no overlapping writes from different nodes, thus it does not need complex data synchronization policies; and 3) the replacement management policy is straightforward (an age-based expiration algorithm; the oldest data will be chosen and dumped to the PFS), not as complicated as in traditional cache systems.

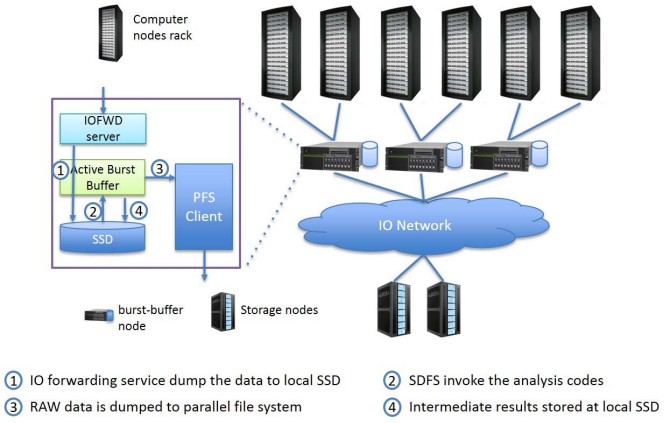


Fig. 1. Architecture of HPC systems with burst-buffer nodes. This diagram highlights the position of Active burst-buffer

The challenges for running analysis codes on burst-buffer nodes are twofold: 1) due to the limited capacity compared to the parallel file system, burst-buffer may not be able to hold all of the generated data locally until the simulation completes. Thus, the analysis codes can only be executed on cached data, which may only be a piece of a file, or a subset of the result. This could make it difficult to program analysis codes which are data-oriented; 2) the cached data in burst-buffer is written by the processes in designated compute nodes, and each process is in charge of its own portion of the file. Thus, the data is distributed across multiple burst-buffer nodes out of order. Therefore, it is important to reorganize the distributed data to the logical files as it will ultimately be presented to the parallel file system, which is an important feature for ease of programming analysis codes. The computing paradigm provided in Active burst-buffer is similar to a MapReduce framework but also differs from it. One of the basic differences is that the data in the burst-buffer is written by compute nodes, and it is difficult to make sure that the file is divided into fixed-size blocks. Therefore, existing Hadoop/HDFS systems may not be a good fit for the analysis on burst-buffer nodes. In addition, scientific computing tends to be growingly data driven, and the data does not always stay permanently in the burst-buffer.

The proposed Active burst-buffer system provides the following functions: 1) following the burst-buffer design, it absorbs the I/O data quickly from designated compute nodes and improves the I/O performance; 2) forwards selected data versions to the low level parallel file system for persistence; 3) constructs a global view of a file for in-transit analysis/processing; 4) supports optimization operations, such as data deduplication and compression.

B. Data Management

Scientific data stored in a file normally has an implicit order. Climate data is a typical example. Climate models simulate the climate data in chronological order, with each process simulating an assigned geographic location (indexed by (lons, lats)). Therefore, the data stored in the file has to follow the same order. Figure 2 illustrates climate data and its mapping to a file. Analysis codes usually have to use this information to complete their tasks. For example, climate scientists usually analyze and compare the data for different areas to get the region of interest. Completing this task requires the max/min/mean temperature/pressure of each month/semester/year for a long time upon given locations. Based on the above climate data model, each analysis process needs to skip the data for other locations to find the related data. At the same time, the analysis results are also kept with a similar order for each location for future visualization. Thus, keeping the data in order is a basic function that an analysis system should provide. A key challenge for the Active burst-buffer system is how to get this information for the analysis code while only caching a piece of data.

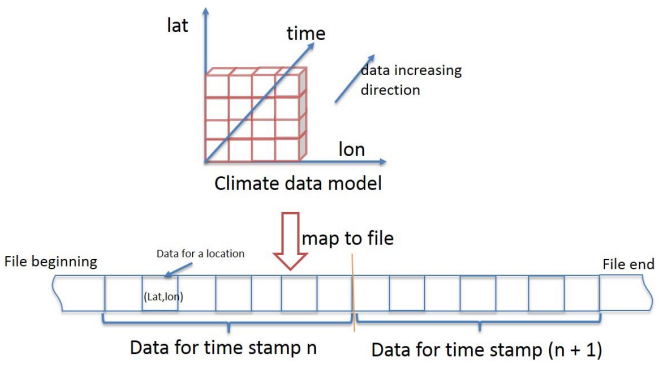


Fig. 2. Illustration of climate data model and its mapping to a file

There are three different I/O patterns that have been observed from HPC applications, N-N, N-m (m $<$ N), and N-1. In an N-N pattern, each process writes the data in a separate file. In an N-m pattern, processes can be considered to be divided into m groups, and each group writes the data to the same file. In an N-1 pattern, all of the processes will write the data to a single file. Active burst-buffer manages the data in file granularity and supports all of these patterns. We take N-1 as an example to illustrate how Active burst-buffer manages data for a file.

Open: When the "open" operations of the simulation are forwarded to the burst-buffer nodes, the following occurs: if the file is new, each burst-buffer node will create a new file in its local storage (e.g. SSD). Otherwise, it will open the file with that name from its local storage.

Write: In an N-1 pattern, each process will write its data to the allocated portion of a file, as illustrated in Figure 3. Each write operation can be abstracted as a triple $<file, offset, size>$. When a write request arrives, the Active burst-buffer will write the data to its local file at the specified offset. At the same time, the Active burst-buffer will record this information for each write operation and each file for the data cached in the burst-buffer, and then use this information to reorganize the data to logical files for the analysis codes. The portions for which there isn't local data will be presented as HOLEs, as shown in Figure 3. When the data is moved to the parallel file system, the related portion will be updated as a HOLE. There are two possible methods to implement this mechanism. The first one is to create an index file using a distributed key-value table to record the HOLE information for each file, which is similar to the method used by PLFS [15]. The other, more efficient, way is to leverage the HOLE functions provided by Linux kernel 3.8 and low-level local file systems (e.g. ext4 and ZFS), which is more efficient. However, it currently supports only block-size HOLES. If the HOLE size is smaller than block size, it will be not identified and the Active burst-buffer will fill in this data with zeroes.

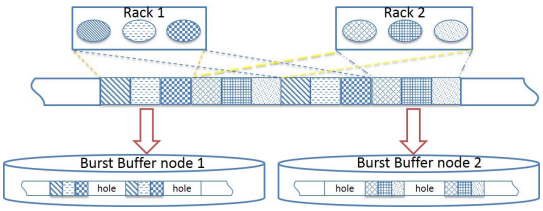


Fig. 3. Active burst-buffer creates a file at each burst-buffer node. The data will be stored at its logic position in the file, for the portion where there is no data, it will be represented as a HOLE. In the figure, each burst-buffer node serves 1 rack of compute nodes. Objects with the same pattern imply there is a mapping between the process and its portion of file.

Read: When a read operation is issued from an analysis code, it is also represented as a triple $<file, offset, size>$. When the Active burst-buffer node receives a read request, it will first check its local storage to get the file. If the local storage does not have the data or only has a portion of data for the request, it will update the request triple with the new $offset$ and $size$, and then will distribute the request to the other burst-buffer nodes. Each burst-buffer node will check its local data and return the related portion of the requested data if there is a match. If the node has no data, it will return zero. The local Active burst-buffer instance, where the read request was received, will aggregate the data as a single I/O response and return it to the analysis process.

C. Interface

The Active burst-buffer is a "file system" that is deployed when the system is built. When it is deployed, it will create a virtual device file "/dev/abbfs" for communication with users. A set of utilities are designed to help users hook their own implemented analysis functions to the Active burst-buffer. The Active burst-buffer allows users to register the analysis functions for each file through *abbfs-register* utility. Active burst-buffer requires the user to compile their analysis functions into a dynamic library. When users register their analysis functions, they need to specify the analysis function name and the path names of related libraries. The other three parameters users have to pass to the *abbfs-register* utility are the file name on which the analysis function is applied, the output file name where the intermediate data should be stored, as well as the registered operation type (map or reduce). *Abbfs-register* will generate a configuration, in the format shown in Figure 4, and write to "/dev/abbfs". When an open request from a simulation application is received, the Active burst-buffer will initialize the system by reading the configuration from "/dev/abbfs" and mapping the analysis operations for each output file of simulations.

Considering special requirements of specific analysis operations, users may choose to analyze the data of multiple writes at a time instead of each one, Active burst-buffer allows users to group simulation writes and cache them in the burst-buffer simultaneously. An extreme example is, if the output size is smaller than burst-buffer cache capacity, users can cache all of data in burst-buffer, and run the analysis application as normal on burst-buffer nodes. Currently, users need to take care of the balance between the I/O size and the burst-buffer capacity.

```
<simulationfile>
    op: func
    lib_path: /users/username/folder/lib.so
    output: outputfile
    cache times: 3
    ......(other options)
</simulationfile>
```

Fig. 4. A configuration file sample and format for communication between Active burst-buffer and users. *Simulationfile* is the name of file where the simulation data will be stored. *Outputfile* is for storing intermediate results.

D. Analysis Framework

Compared to the applications' outputs, the burst-buffer nodes are only able to cache a piece of file data locally in a fixed amount of time. To rectify this, we designed a MapReduce-like "accumulative computing" framework for analysis codes. The "map" phase is for analyzing raw data

to get intermediate results. The raw data can be dumped to a parallel file system for later use or can be deleted if users do not need it any more. The "reduce" phase is for aggregating intermediate results to get the final result. Active burst-buffer requires that the analysis operations significantly reduce the data size (the resulting size should be small enough to be stored at burst-buffer), such as max, min, and histogram operations. Operations that cannot reduce the data size, such as sorting, may not benefit from Active burst-buffer depending on the data set size and the burst-buffer's local storage capacity. Users can still use burst-buffer to dump out data from compute nodes quickly, or use burst-buffer for local sorting and storing the results into a parallel file system for further global sorting. Active burst-buffer cannot reduce the data movement in these scenarios but can still be useful for in-transit processing.

Active burst-buffer supports two types of "Map" functions: embarrassingly parallel operations, e.g. filter (threshold), and non-embarrassingly parallel operations, such as extracting a max/mean/min temperature of each month/semester/year from a data set containing years of climate data. Active burst-buffer is designed with a basic assumption that users have the knowledge of data organization and data type of simulation output in each I/O cycle. For example, if the output of the simulation is a **double** array, the user cannot consider it as **int** array by mistake. Users have to follow strict interface for programming analysis codes. For the embarrassingly parallel case, the interface is defined as:

***void** func(**void** *buf, **size_t** size, **int** fd)*

and the Active burst-buffer will feed the data of each simulation, as well as intermediate file description (fd) to the registered analysis function. Users will process the data, and explicitly write the result to the intermediate file (fd). For non-embarrassingly parallel operations, the interface is defined as:

***void** func(**int** in_fd, **int** out_fd, **MPI_Comm** comm)*

and the Active burst-buffer allows the users control everything. Active burst-buffer will create an MPI process at each buffer node, and initialize the MPI environment for each process with the help of hydra process management. Each MPI process will call user-implemented functions for analysis, and the user needs to divide and assign the tasks to each rank, as writing a normal analysis code with reading the data from files explicitly. When the required data has not been yet produced by the simulation, the Active burst-buffer will block the read operation until the data is ready. The same mechanism is also used for "Reduce" operations.

IV. Evaluation

A. Prototyping

To evaluate our idea, we have built an Active burst-buffer prototype using FUSE library. In our existing prototype, we used the HOLE mechanisms provided by Linux Kernel 3.8 and Ext4 file systems. We configured the block size of the Ext4 file system as 1KB, to make the HOLES as fine-grained as possible and to improve the system performance. For the HOLES whose size is smaller than the block size, Active burst-buffer will read it as usual, and then uses a logical OR operation (a hole is filled with 0s by Ext4) in the read process to combine the data of each response from different burst-buffer nodes, and thereby retaining the correct data. We implemented the data management component, and the complete computing paradigm is ongoing. In the following experiments, we embedded and precompiled analysis codes in the file management component. As a result, our experiments do not benefit from overlapping the compute phases and I/O phases as designed, which should be an advantage of Active burst-buffer. Even with this artificial restriction, our experimental results are still promising, which shows the promise of our approach.

B. Experimental platform and Methodology

We evaluated the proposed Active burst-buffer concept and the prototype on the Kodiak cluster. Kodiak is a part of the Parallel Reconfigurable Observational Environment (PRObE) project [16] which aims to provide large-scale system research platform. The Kodiak system has 1024 nodes, and each node is equipped with two sockets of Single Core AMD Opteron 252 (2.6GHz, 64bit) CPU and 4GB memory. The platform provides both single data rate (SDR) Infiniband interconnect and a standard Gigabit ethernet network.

In our experiments, we built a cluster testbed with totally 245 nodes to emulate the system architectures. We mimicked the evaluated schemes by customizing the software stacks. We used 5 nodes as storage nodes to provide a total of 5 TB of storage capacity. Parallel virtual file system (PVFS2) is deployed to manage storage nodes and to provide high performance I/O operations. The measured peak bandwidth in our configuration is 352MB/sec. From the remaining nodes, we dedicated 24 as a analysis cluster, and each node is configured as a PVFS2 client. The rest of nodes are configured as compute nodes and burst-buffer nodes. The number of compute nodes and burst-buffer nodes varied according to experiments' requirements. I/O forwarding scalable layer (IOFSL) middleware is deployedr for communication between compute nodes and burst-buffer nodes or simple I/O forwarding nodes. This represents a realistic scenario in large scale supercomputers.

1) Evaluated Scenarios: We compared the performance of Active burst-buffer with three normal scenarios, which are widely used in existing frameworks. We compared their performance under different ratios of compute nodes and burst-buffer nodes (including 8:1, 16:1 and 32:1), different scales of compute nodes (scaled from 16 compute nodes to 192 compute nodes), and different application output size (from 64GB to 1TB). The details for each scenario are introduced as follows (in the following texts, ANALY represents analyzing the simulation output at a dedicated analysis cluster after simulation is finished):

1) PFS+ANALY: PFS, short for parallel file system. This scenario emulates a basic Beowulf cluster architecture that contains compute nodes and storage nodes, as illustrated in

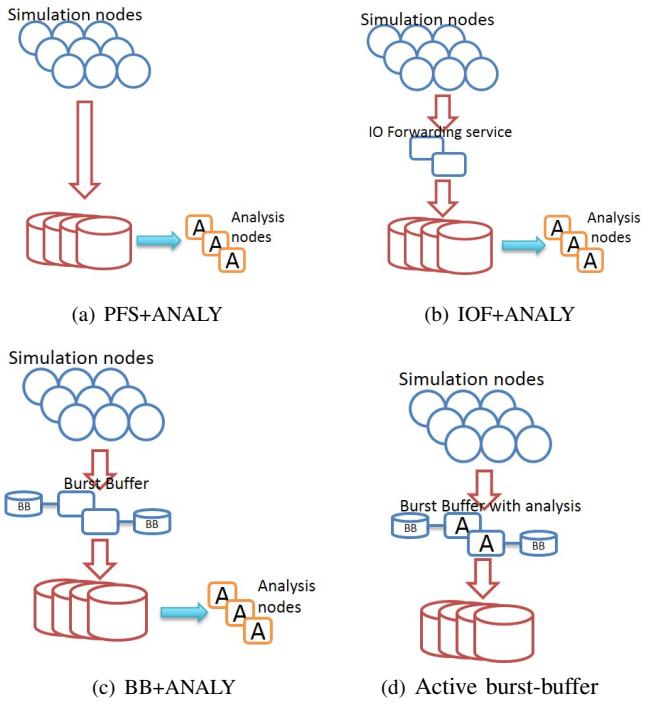


Fig. 5. Evaluated Scenarios

Figure 5(a). Data is analyzed offline on a separate analysis cluster when the simulation is finished.

2) IOF+ANALY: IOF, short for I/O forwarding. This scenario emulates the Blue Gene/L (Intrepid) architecture, that has an I/O forwarding layer. Compute nodes will forward their I/O operations to I/O nodes through the I/O forwarding middleware (e.g. IOFSL in our experiments), as illustrated in Figure 5(b). After data is written to the parallel file system by the simulation, a separate cluster will read the data from the parallel file system for analysis.
3) BB+ANALY: BB, short for burst-buffer. This scenario emulates the burst-buffer architecture, as illustrated in Figure 5(c). It is similar to IOF, but each I/O node (we call burst-buffer node) is enhanced with burst-buffer as quick temporary storage. When compute nodes forward their write operations to burst-buffer nodes, burst-buffer nodes will temporally store the data in burst-buffer, let applications return to computing as soon as possible. When the simulation is completed, a separate analysis cluster will read the data from parallel file system for analysis.
4) ABB: short for Active burst-buffer, the scenario designed in this study, as illustrated in Figure 5(d). In this workflow, The data will be cached and analyzed at the burst-buffer before it is moved to the parallel file system. The intermediate analysis data will be stored at burst-buffer, then when the simulation is complete, the intermediate data will be aggregated to get the final analysis results.

Concisely, the evaluated the scenarios are based on three typical architectures, BB and ABB share the same architecture. And in the rest of paper, we will use PFS, IOF, BB and ABB to represent them respectively.

2) Application: In experiments we used the IOR benchmark, combined with various analysis codes, as our applications running on the compute nodes. We selected the IOR benchmark based on two considerations: 1). it is simple and easy for us to control the output size; it is also straightforward for us to compose the related analysis kernels without much effort in understanding the output structure of real applications, which is not the main work of this study; 2). such evaluated scenarios are completely data-intensive application scenarios, and there is almost no computing phase. Therefore, it can help show the costs of data movements and the efficiency of our Active burst-buffer in reducing data movements.

3) Analysis codes: We evaluated the Active burst-buffer with three normally used operators, including filter, max, and average, from Climate Data Operators (CDO) library [17]. We implemented each of them for the Active burst-buffer and the other compared scenarios, as described in the following. For dedicated analysis clusters we implemented the analysis code with MPI. For Active burst-buffer there is a different implementation described below.

1) ***Filter***: filter is an embarrassing parallel code, it acts like a threshold function, and helps scientists select interesting data. We implemented it as a function for Active burst-buffer and pre-compiled it in Active burst-buffer code.
2) ***Max***: returns the max value from the whole data set. For the Active burst-buffer, we implemented it in two separate parts. The first part, precompiled in the Active burst-buffer, is to calculate the local max value for each simulation I/O, which selects the max from the local burst-buffer and keeps the intermediate value locally. The second part processes the local max values to get the final global max value, when the simulation is finished.
3) ***Average***: get a the mean value from the entire data set. We implemented it in the same way as the max operation.

C. Evaluation Results and Analysis

Before the presentation of our experimental results, we need to state that, in each experiment, the number of dedicated analysis nodes is the same with respect to the number of burst-buffer nodes used in the Active burst-buffer scenario to make sure that the same computing resources are used for analysis in different scenarios for fairness. For example, in a experiment with 128 compute nodes and 16 burst-buffer nodes, the dedicated analysis cluster also has 16 nodes. In our experiment, we assume that dedicated analysis cluster shares the same storage with the simulation cluster, although, in practice, they somtimes have separate storage system, which will are more costly for data movements. Since the ratio between the compute nodes and burst-buffer nodes has a great impact on workloads of each burst-buffer node (or I/O node) for IOF, BB, and ABB scenarios, it will have a great impact on the performance of these scenarios. Therefore, we evaluated the performance of Active burst-buffer with three typical ratios: $8/1$, $16/1$, and $32/1$.

In the rest of paper, we use $r/1$ style to denote the ratio configuration between compute nodes and burst-buffer nodes

and use $n:m$ style to denote the scale of systems. Specially, n represents the number of computer nodes, sometimes we simply use it to represent system scale; and m represents the number of burst-buffer nodes, and it is determined by n and r. We include it in the figures for easy to read.

1) Baseline experiments: In this section, we present the I/O bandwidth of each emulated scenario, which can help us to get a brief understanding about the characteristics of each emulated case. Figure 6 presents the bandwidth of IOF, BB, and PFS scenarios under different scales. The ratio between the compute nodes and burst-buffer nodes for each experiment in the figure is configured as $8/1$. As stated in the figure, I/O forwarding has worse performance than the other two scenarios. In each experiment, its I/O time (not including analysis time) is much worse than BB and PFS (only I/O) scenarios, which is unexpected. Two of the main reasons that caused this situation are due to not having a lightweight operating systems tuned for IO forwarding like those used in existing machines, and not enough scale to cause serious contention on the parallel file system for the PFS scenario. Therefore in the following experiments we skipped the IOF configuration and mainly compare the Active burst-buffer against BB and PFS workflows.

In addition, another thing we observed from the figure is that BB has poor I/O bandwidth compared to PFS at small scale and achieves better bandwidth than PFS in large scale (e.g. 128 in our experiment for $8/1$ configuration). It shares similar reasons with IOF for skewed performance. And with help of a local "cache" functionality, it performs a bit better than IOF when the system is scaled to 128 nodes in our experiments. Meanwhile, as we change the ratio between compute nodes and burst-buffer nodes to $16/1$ and $32/1$, we find that BB performs worse than PFS in all evaluated scales, shown in figure 7. However, we can observe that, in each configuration, the I/O bandwidth of BB is increasing steadily, while the I/O bandwidth of PFS is decreasing steadily with increasing scales. (Because the ratio configuration has no relationship with PFS, we can observe this trend from the whole figure.)

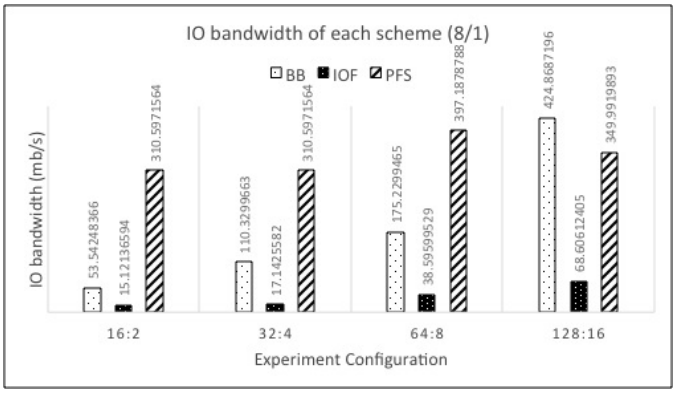


Fig. 6. IO bandwidth of each configuration evaluated in our experiments with $8/1$ configuration.

2) General Performance: In this section, we present the general performance of Active burst-buffer , BB+ANALY and PFS+ANALY scenarios. We used three analysis codes, mentioned previously, to evaluate the performance of these three scenarios under three typical ratio configurations, different scales and data volume size. In this section, a task contains not only simulation workloads, but also the output data analyzed with the given analysis operations.

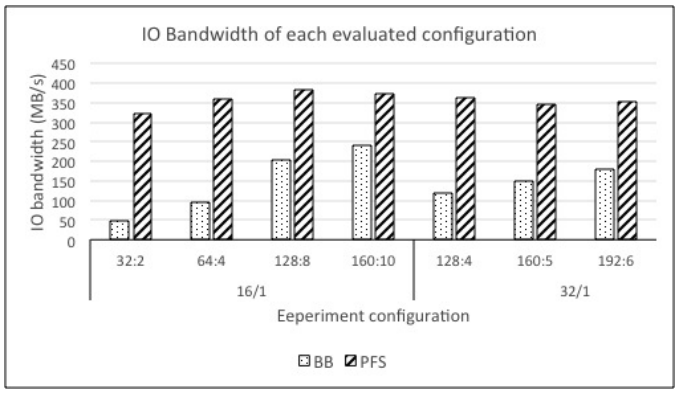


Fig. 7. I/O Bandwidth of each scenario under $16/1$ configuration and $32/1$ configurations. The left part of the figure is for $16/1$ configuration, and the right part of the figure is for $32/1$ configuration.

8/1 configuration: Figure 8 presents the execution time of of evaluated operations under different scales (including nodes and data size). the ratio between compute nodes and burst-buffer nodes is $8/1$. As printed in the figure, Active burst-buffer always has the shortest execution time. Compared with the BB+ANALY scenario, it saves execution time by around 45% in average, and more than 55% for some specific configuration (e.g. system scale at 128 nodes) and analysis operations. Compared with the PFS+ANALY scenario, it saves total execution time by more than 40% for all of operations, and this saving is increasing steadily while the system is scaled to a larger number of nodes. This is because, when the system scales, the many processes simultaneously accessing files will cause a serious performance degradation on the parallel file system and make data movements more expensive.

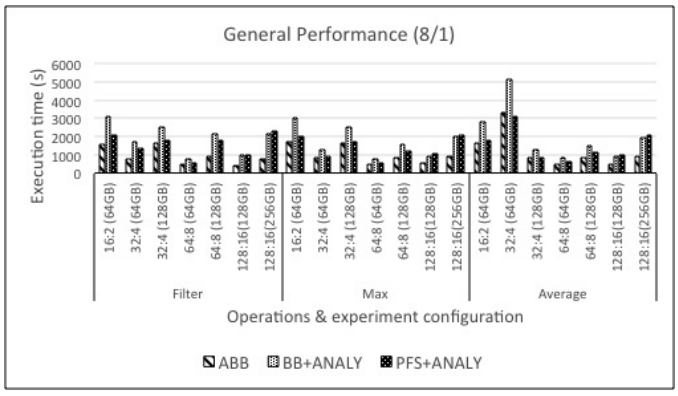


Fig. 8. Execution time of each scenario with different system scale and data size with $8/1$ ratio. The left part of the figure is for filter, the middle part is for max, and the right part is for average .

16/1 configuration: Figure 9 presents the performance

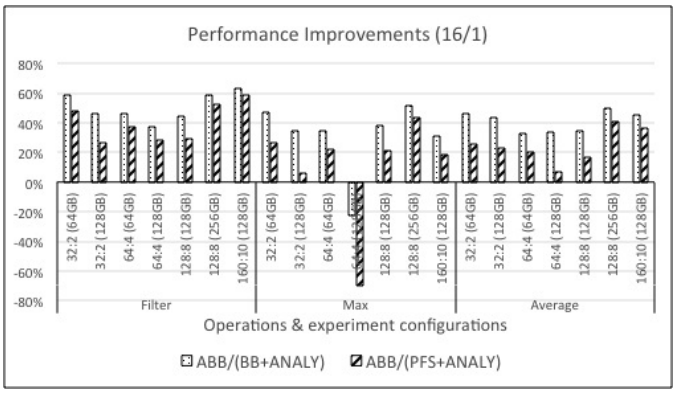


Fig. 9. Performance improvements of Active burst-buffer different system scale and data size with $16/1$ ratio. The left part of the figure is for filter, the middle part is for max, and the right part is for average.

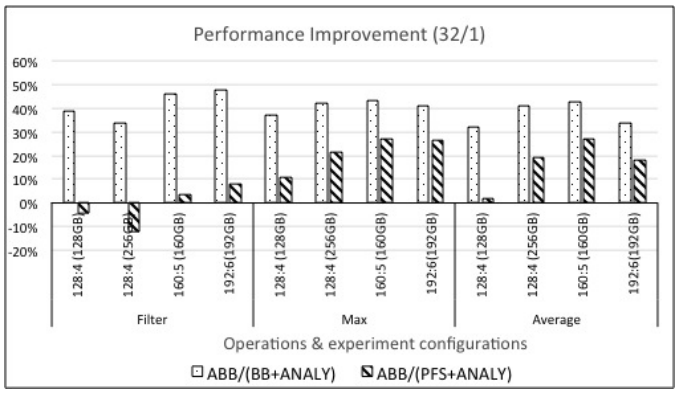


Fig. 10. Performance improvements of Active burst-buffer with $32/1$ ratio. The left part of the figure is for filter, the middle part is for max, and the right part is for average.

improvements of ABB against BB+ANALY and PFS+ANALY when the ratio is configured as $16/1$. As the ratio increases by 2x, the workloads of each burst-buffer node theoretically increases by 2x also, which means less resources can be used for analysis codes. However, for most cases we still observed 30% in performance improvements in average.

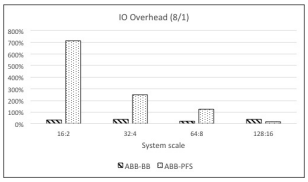
32/1 configuration: Figure 10 presents performance improvement of ABB with $32/1$ configuration. Compared to BB+ANALY, there is around 35% performance improvements in average. However, compared to PFS+ANALY, we observed performance degradation of Active burst-buffer for filter operations when the scale is 128 nodes. the degradation is mainly caused by incompatible hardware configuration. For the Active burst-buffer, the ratio between the compute nodes and burst-buffer nodes is 32/1, which is smaller than the ratio between the compute nodes and storage nodes (128/5) used in PFS+ANALY. Therefore, When the system scale reaches 160 nodes, it still performs better than PFS+ANALY scenario, with 8% improvements.

In conclusion, at reasonable scales, Active burst-buffer can perform better than existing scenarios. The ratio between compute nodes and burst-buffer nodes has a great impact on the system performance. From the experiments, the typical 8/1 or 16/1 ratio is suggested for configurations.

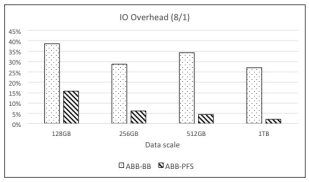
D. Overhead of Active burst-buffer on I/O operations

In this section, we analyze the overhead of Active burst-buffer. Active burst-buffer introduces the in-transit analysis at burst-buffer nodes, and the data movement will be delayed until it is processed by the analysis codes. We define this delay as the overhead of Active burst-buffer on I/O operations. In our experiments, due to there being no overlap between the compute phase and the I/O phase in our prototype, this overhead will be directly reflected on IOR's execution times. Also, compared to the BB scenario, the Active burst-buffer adds an analysis capability. Therefore, the overhead of Active burst-buffer to BB scenario can be defined as $T_{overhead} = T_{ABB} - T_{BB}$, where T_{ABB} is the execution time of IOR under Active burst-buffer scenario, and T_{BB} is the execution time of IOR under BB scenario (without analysis). We also used the similar method to define the overhead of Active burst-buffer to PFS scenario as $T_{overhead} = T_{ABB} - T_{PFS}$. In these experiments, we fixed the data size and varied the system scales to see how the system scale will affect the overhead, and fixed the system scale and varied data size to see how data size will affect the overhead.

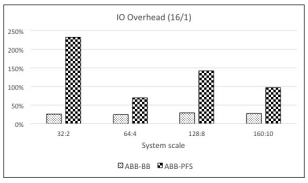
Figure 11 plots the overhead of Active burst-buffer compared to BB and PFS scenarios under 8/1 and 16/1 ratio configurations respectively. Compared to BB scenario, Active burst-buffer holds around $30\% \sim 40\%$ delays of simulation completion time steadily. However, compared the PFS scenario, the overhead is considerable, and there is around 700% overhead when there is 16 nodes with $8/1$ ratio configuration. However, from the figure, we can also find that the overhead is obviously decreasing while the system scale or data volumes increase. When the system scales to 128 nodes or data size is increased to 1TB, the overhead of Active burst-buffer to PFS scenario is decreased to no more than 10%.



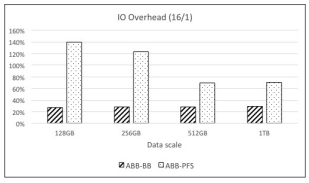
(a) $8/1$ with fixed 128GB data



(b) $8/1$ with fixed 128 nodes



(c) $16/1$ with fixed 128GB data



(d) $16/1$ with fixed 128 nodes

Fig. 11. I/O overhead under different data scale and system scale. 11(a) and 11(b) with $8/1$ configuration. 11(c) and 11(d) with $16/1$ configuration.

E. Data Movement Costs

In this section, we evaluate the costs of data movements for offline analysis. We use I/O time to represent data movements and assume that the analysis application logically contains two workloads: 1). Load the data from the storage system (I/O); 2). Processing the data (computing) to get results. An easy way to study the costs of data movements for analysis codes is to compare its I/O time against a reference. In this subsection, we compare the I/O time of analysis codes against total execution time of each scenarios, including simulation and analysis.

In these experiments, we fixed the system scale as 128 nodes and vary the data size from 128GB to 1TB. We evaluated with $8/1$ and $16/1$ ratio configurations. Figure 12 compares the I/O time of filter analysis code to the total execution time of BB and PFS scenarios. From the figure, we found that the costs of data movements involved in the analysis application are considerable. In the $8/1$ configuration, more than 50% of the time is spent on read operations for analysis codes. In the $16/1$ configuration, the data movement costs are reduced to 30%, but there are still considerable costs.

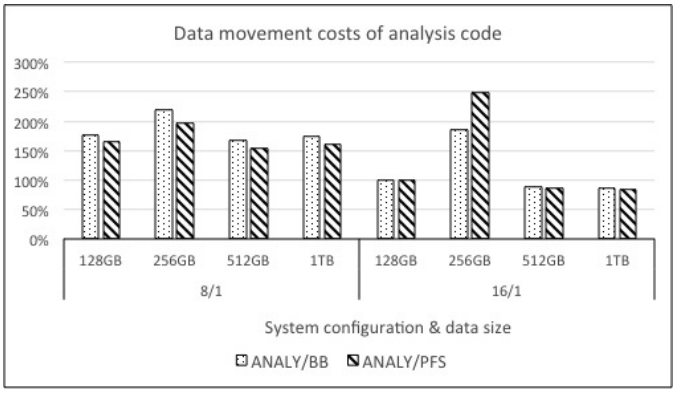


Fig. 12. Data movement involved in analysis compared to I/O time of simulation application.

V. Related Work

Various solutions have been proposed for I/O bottleneck issues in HPC systems. In this section, we review and discuss numerous existing studies related to our study.

A. Hybrid Storage and Burst-buffer

Extensive studies [18, 19] have focused on integrating SSDs into HPC systems. Hystor [20] is an exemplar hybrid storage system where hot data (performance critical blocks) is placed in SSD whereas other cold data is held in conventional HDD. Among these studies, burst-buffer [8, 9] has been proven as a promising solution for addressing the I/O bottleneck issue for data-intensive HPC applications. It utilizes Solid State Drives for high bandwidth and overlaps the computing phase and I/O phase of applications. Our work is designed based on the burst-buffer and is an extension/enhancement to existing burst-buffer concept and design.

B. Active Storage and Active Disk/Flash

Active storage [6, 21–23] explores the computing resource at storage nodes for analysis, and specially fit read-intensive operations. It deploys the analysis code directly on storage nodes where the data is located. For each read operation, it will invoke the analysis code and only return small-size results to the application. Normally, the analysis code is a part of an application, which requires code changes for applications. In contrast, our design is decoupled from the application, and applications do not need to drive the analysis. Also, while active storage invokes the analysis code at the read phase, our work invokes the analysis code at the write phase. At last, considering the storage servers are shared by a global system, it has limited computing and memory resources for analysis codes, compared to burst-buffer nodes that serve dedicated nodes. In other words, the Active burst-buffer framework can also remove the raw data directly after obtaining the analysis results, if users do not want to store the raw data, which can reduce huge amounts of data movements.

Active Disks [24, 25] share a similar idea with active storage except exploiting the computing resource embedded in disks. Recently, accompanied with the emergence of Solid State Drives, the idea of Active Disk is being applied to SSD [26]. However, as pointed in [26], existing SSDs can not conduct complex analysis operations and have limited performance and flexibility due to performance of embedded processor unit.

C. In-transit/In-situ Analysis

Recently, in-transit/in-situ analysis attracted intensive attentions [14, 27, 28]. PreDaTA [14] is a typical example. It utilizes RDMA technology to retrieve the data to dedicated staging nodes for analysis. Our study shares a similar idea with PreDaTA, but with several differences. First, our work explores the computation resources on burst-buffer nodes that are integrated in the system, instead of additional dedicated nodes. Second, in PreDaTA, each staging node needs to aggregate all of the data from each served compute nodes before analyzing operations. However, our work does not have such constraints. In addition, because we can reorganize the data into a logical file, the analysis code in our platform could be more flexible and powerful.

VI. Conclusion and Future Work

The explosion of scientific data volume undoubtedly has brought significant challenges to the HPC research and development community. Reducing the data movement is a key for both performance improvement and energy saving for future HPC systems. Many recent studies approach this problem and try to combat the data movement issue including the widely recognized burst-buffer concept and solution.

In this study, we extend the burst-buffer concept and propose a new Active burst-buffer concept and design. Active burst-buffer not only makes scientific simulations able to dump out data quickly and overlap the computing and I/O phases, but more importantly explores the computing resources on burst-buffer nodes and conduct in-transit analysis on cached data

before written to the storage system. We have designed and prototyped the Active burst-buffer framework including a file system based on FUSE to reorganize the cached data to a logical file and a MapReduce-like computing paradigm for conducting analysis codes. We carried out extensive evaluations on the PRObE testbed. The current results have shown substantial performance gain and are very encouraging, which confirmed that Active burst-buffer is capable of enabling data analysis in-place on burst-buffer nodes and is a promising solution to scientific discoveries with growing data demands. We hope to share this new Active burst-buffer concept and research and development findings with the community through this paper and call for the community's collective efforts to further explore these ideas and build production systems following these ideas to combat the increasingly critical data movement issue. Such efforts are needed to build future HPC systems.

VII. Acknowledgments

This work was performed at the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory, supported by the U.S. Department of Energy contract DEFC02- 06ER25750. The publication has been assigned the LANL identifier LA-UR-13-27637. This research is also sponsored in part by the National Science Foundation under grant CNS-1162488.

References

[1] "Community Earth System Model," http://www.cesm.ucar.edu.

[2] "Gyrokinetic Particle Simulations Gyrokinetic Toroidal Code," http://w3.pppl.gov/theory/proj_gksim.html.

[3] A. Szalay, "Extreme Data-Intensive Scientific Computing," *Computing in Science Engineering*, vol. 13, no. 6, nov.-dec. 2011.

[4] "I/O Forwading Scalability Layer," www.iofsl.org/.

[5] " DOE Extreme-Scale Technology Acceleration—FastForward," https://asc.llnl.gov/fastforward/.

[6] S. Woo, S. Samuel, L. Philip, C. Robert, and R. Rajeev, "Enabling Active Storage on Parallel I/O Software Stacks," in *MSST'10*, 2010.

[7] C. Chen, Y. Chen, and P. C. Roth, "DOSAS: Mitigating the Resource Contention in Active Storage Systems," in *Cluster'12*, 2012.

[8] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *MSST'12*, 2012.

[9] J. Bent, S. Faibish, U. Gupta, G. Ma, J. Pedone, P. Tzelnic, J. A. H. Chen, G. Grider, J. Patchett, and J. Woodring, "Co-processing simulation and visualization on a prototype exascale burst buffer storage system," 2011.

[10] G. Grider, "Exascale fsio/storage/viz/data analysis can we get there? can we afford to?" 2011.

[11] C. Chen and Y. Chen, "Dynamic active storage for high performance," in *ICPP'12*, 2012.

[12] Y. Kim, R. Gunasekaran, G. Shipman, D. Dillow, Z. Zhang, and B. Settlemyer, "Workload characterization of a leadership class storage cluster," in *PDSW'10*, 2010.

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," in *MSST'11*, 2011.

[14] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "Predata–preparatory data analytics on peta-scale machines," in *IPDPS'10*, 2010.

[15] J. Bent, B. Mcclelland, G. Gibson, P. Nowoczynski, G. Grider, J. Nunez, M. Polte, and M. Wingate, "Plfs: A Checkpoint Filesystem for Parallel Applications," Tech. Rep., 2009.

[16] G. Grider, "Parallel Reconfigurable Observational Environment (PRObE)," Oct. 2012. [Online]. Available: http://www.nmc-probe.org

[17] U. Schulzweida. (2014) Cdo users guide. [Online]. Available: https://code.zmaw.de/projects/cdo/embedded/1.6.3/cdo.html

[18] A. Badam and V. S. Pai, "Ssdalloc: Hybrid ssd/ram memory management made easy," in *NSDI'11*, 2011.

[19] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *ISCA'08*, 2008.

[20] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," in *ICS*, 2011.

[21] Y. Xie, K.-K. Muniswamy-Reddy, and D. F. etc., "Design and Evaluation of Oasis: An Active Storage Framework based on T10 OSD Standard," in *MSST*, 2011.

[22] B. G. Fitch, A. Rayshubskiy, M. P. T.J. Chris Ward, B. Metzler, H. J. Schick, B. Krill, P. Morjan, and R. S. Germain, "Blue Gene Active Storage," in *HEC FSIO R&D Workshop '10*, 2010.

[23] C. Chen, M. Lang, and Y. Chen, "Multilevel active storage for big data applications in high performance computing," in *Big Data'13*, 2013.

[24] G. Chockler and D. Malkhi, "Active Disk Paxos with Infinitely Many Processes," in *PODC'05*, vol. 18, 2005.

[25] S. Chiu, W.-k. Liao, and A. Choudhary, "Design and Evaluation of Distributed Smart Disk Architecture for I/O-intensive Workloads," in *Proceedings of the 2003 International Conference on Computational Science*, 2003.

[26] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *MSST'13*, 2013.

[27] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *SC'13*, 2013.

[28] J. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC'12*, 2012.