Cloud-based Burst Buffers for I/O Acceleration (Unrefereed Workshop Manuscript)

Tianqi Xu[1] Kento Sato[2] Satoshi Matsuoka[1]

Abstract: Cloud computing offers high computational resources, scalability, as well as ease of access. Such cloud environments provide users with virtually unlimited computational resources to run HPC applications at larger scale than what in-house systems can provide. Since large scale data intensive applications typically generate huge amounts of intermediate data and are shared by hundreds and thousands of compute nodes, such applications require high I/O throughput to shared storage. However, current shared storage in cloud environments cannot provide enough I/O throughput for these applications. The low I/O throughput becomes a performance bottleneck and the prolonged execution time incurs more cost to users as most cloud providers employ pay-as-you-go pricing models. Furthermore, the eventual consistency policy adopted by most cloud storages causes multiple-node job failure due to the inconsistent read-after-write.
To solve these problems, we propose a cloud-based burst buffer system as a new tier in cloud storage systems. The cloud-based burst buffer system uses computing nodes as burst buffer nodes, and buffers applications' data in the burst buffer nodes. Because throughput between compute nodes is much higher and more stable than shared storage throughput, we can accelerate I/O performance for data intensive applications. Moreover, by maintaining data consistency among burst buffer nodes, we can avoid job failure caused by eventual consistency issue. To explore the effectiveness of cloud-based burst buffers, we implement a prototype and evaluate the system in Amazon EC2/S3. Our experiments reveal that our system can perfectly solve the eventual consistency issue as well as improve performance of a real-world data intensive application by up to 4.5 times as well as reduced monetary cost by 56.3%.

Keywords: cloud computing, burst buffer, data intensive applications

1. Introduction

Cloud computing architecture has been gathering the attention of application developers because of its elasticity. With the elasticity, users can enjoy virtually unlimited computational resources on the fly and pay a cost depending on their usage. In addition, recent clouds also provide computational resources for high performance computing (HPC) [1–5]. For example, Amazon EC2 provides HPC instances with high bandwidth networks, I/O subsystems with high performance SSDs, and accelerators such as GPUs [6]. These characteristics make cloud computing more attractive for large-scale scientific applications.

Network and storage bandwidth are, however, still insufficient for highly data intensive HPC applications such as big data analysis and large scale visualization. For example, Amazon Simple Storage Service (Amazon S3), a cloud-based shared storage, provides only a few hundreds MB/s of throughput, whereas parallel filesystems of supercomputers can serve up to a Terabyte per second. This is because instances in Amazon EC2 are connected to Amazon S3 via a slower IDC networks and it is shared among many users. Due to growing demands for higher I/O throughput, current cloud environments will not be able to provide sufficient I/O performance for such HPC applications. Furthermore, due to the eventual consistency policy typically facilitated by common cloud storage systems such as Amazon S3, multi-node workflow applications may fail because of inconsistent reads [7, 8].

To solve these problems, we propose a cloud-based burst buffer system. Burst buffers have been proposed as a new tier of storage hierarchy in supercomputers, providing higher I/O throughput for shared storage by absorbing bursty I/O requests from applications [9]. Moreover, because most HPC applications exhibit data access locality, if we cache the data, which will be accessed in the near future on the burst buffers, we can accelerate access performance [10]. Our idea is to build on-demand burst buffers in the cloud instead of statically allocated I/O nodes used in supercomputers by taking advantage of the cloud's elasticity. In order to achieve high scalability as well as monitor and balance the workload, we propose a new hybrid Master/Client, Key/Value model for our cloud-based burst buffer system. In addition, by maintaining data consistency among burst buffer nodes, we can avoid the eventual consistency problem. To explore the

[1] Tokyo Institute of Technology, Japan, Tokyo, Meguro, okayama 2-12-1
[2] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551 USA

effectiveness of burst buffers in clouds, we implement our proposed system and measure the I/O performance and the execution time of two real scientific data intensive applications, Montage [10] and Supernovae, on a public cloud environment, Amazon EC2/S3. Our experimental results using Amazon EC2/S3 show that we achieve 10.47 times improvement in read throughput and 16.8 times improvement in write throughput with a burst buffer system with 8 I/O nodes. Additionally, our system scales well by increasing the number of masters, and achieves up to 2.5 times performance improvement in meta data operations compared to Amazon S3. The executions of Montage and Supernovae show that we can improve the performance of real data intensive applications by 4.5x as well as save 2.29x monetary cost. Moreover, by using our system we perfectly avoid the eventual consistency issue which causes frequent job failures on Amazon S3. To the best of our knowledge, this work is the first exploration of applying burst buffer technologies to clouds, and delivering quantitative evaluations.

Our contributions can be summarized as followings:

- A cloud-based burst buffer model as a new tier of storage hierarchy in Clouds;
- A hybrid Master/Client, Key/Value model for scalable burst buffers;
- An implementation of the proposed burst buffer system;
- Evaluations on the proposed burst buffer system with a real data intensive application, Montage, and demonstration of significant speed up.

The rest of this paper is organized as follows. In Section 2, we describe the background and motivation. Then, we describe the overview of our cloud-based burst buffer system architecture in Section 3, and the details of its implementation in Section 4. Next, in Section 5, we present our experimental results of the cloud-based burst buffer system. Finally, we describe related work in Section 6 and conclude in Section 7.

2. Background and Motivation

Executing data intensive HPC workloads in clouds may result in unacceptable performance degradation due to low and unstable I/O performance as well as inefficient file metadata operations in cloud storages, moreover the eventual consistency policy adopted in cloud storages may cause job failure due to inconsistent read (Section 2.1). Our analysis of data intensive workloads on HPC systems also found that a number of applications have temporal I/O locality (Section 2.2). These I/O workloads can be executed correctly and accelerated even on clouds even if performance of shared storage is low and unstable or adopted eventual consistency policy (Section 2.3) by using our cloud-based burst buffer system as an *on-demand remote cache space*.

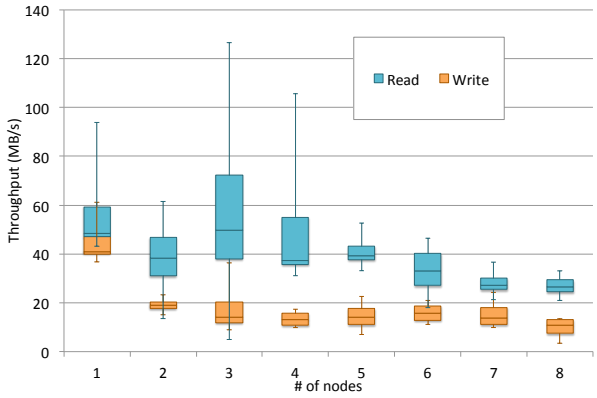

Fig. 1 Read/write I/O throughput with a single shared file on Amazon s3.

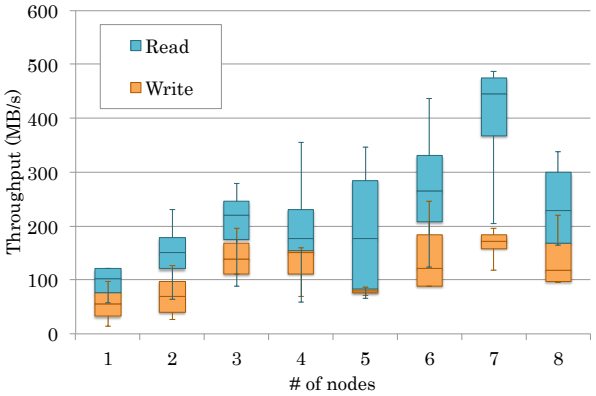

Fig. 2 Read/write I/O throughput with each nodes accessing its own file on Amazon s3.

2.1 Problems in Cloud Storage

The computational power in clouds enables users to run high performance scientific applications faster than ever, which has been making cloud computing attractive to large-scale scientific applications. However if we run multiple-nodes data intensive workloads, which read and write a huge amount of data, we are facing two major challenges: first, the prolonged execution time can be unacceptable because of the low and unstable I/O throughput as well as low file meta-data operations in cloud storage; second, the eventual consistency policy can causes the job failure.

First, we measure the I/O performance on Amazon S3 to illustrate the performance issues we mentioned previously. Fig. 1 shows I/O throughput with N number of nodes concurrently reading or writing to a single file on the Amazon S3 cloud. In the best case, i.e., reading using 3 nodes, the maximum I/O throughput is only 150 MB/s. In a different experiment with N compute nodes accessing to its own individual file, we see improvements in I/O throughput and scalability as shown in Fig. 2. However, the improvements are still limited compared to state-of-the art parallel file systems, especially where write performance does not scale with the number of nodes. From the figures, we also see that the I/O performance is fairly unstable. Because typical data intensive applications consist of processes with mutual dependencies, prolonged

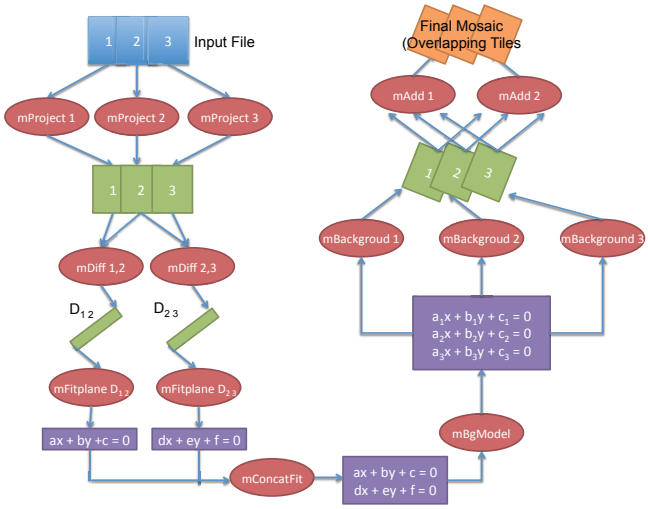

Fig. 3 The Process of Montage [10]

I/O operation due to this instability can propagate, degrading the overall performance [10]. Also, not only I/O throughput but also metadata operations, such as file creation or get file status etc., can become bottlenecks of I/O performance. Hence from these experiments, we find that the I/O performance in cloud storages is insufficient for data intensive HPC applications.

Furthermore, modern clouds provide NoSQL storage substrates such as Amazon SimpleDB [11], HP Helion Object Storage service [12] and Rackspace Cloud Files [13]; such storages typically sacrifice consistency in favor of improved latency for performance, as well as availability, using relaxed consistency protocols such as *eventual consistency* [7]. Although they are fine for standard cloud workloads, for workflows of data intensive HPC workloads, where the result from the preceding nodes in the workflow is passed onto the succeeding nodes under the assumption that there is a high-performance and consistent filesystem such as Lustre, execution in a cloud environment could result in an error as stale data may be read [8], and so far the solutions have been to employ stricter consistency models which sacrifice performance.

These problems have been reported in various studies [14–18]. As such, we need new methodologies for achieving high performance in data intensive HPC workflows in clouds, and our cloud burst buffer is designed exactly to cope with the problem.

2.2 Temporal I/O Locality in data intensive Workloads

Our analysis of data intensive workloads on HPC systems shows that a number of applications have temporal I/O locality. Fig. 3 shows the process of a real scientific workflow application, Montage – a portable software toolkit for constructing custom, science-grade mosaics by composing multiple astronomical images [10]. We can see that the whole process can be divided into several sub processes; each sub-process reads the output of previous sub-processes and generates data for successive

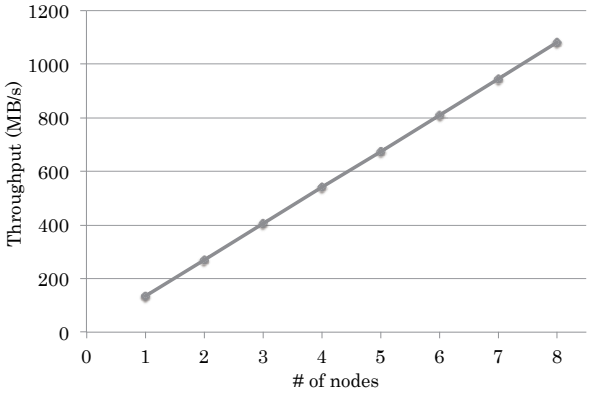

Fig. 4 Point-to-point communication throughput in Amazon EC2

sub-processes. We found that such a process pattern increases the temporal I/O locality of workflow applications.

To investigate the temporal I/O locality of data intensive workloads, we developed MUSE [19] to trace all I/O operations. We used MUSE to determine the I/O locality of Montage with three different datasets. The results of these experiments are shown in Table. 1. The buffered I/O ratio, R_{buf}, means the ratio of read and write operations. The ratio can be formulated as:

$$R_{buf} = \frac{r_b + w_b}{r_t + w_t} \quad (1)$$

where r_t and w_t denote total read and write sizes, respectively; and r_b and w_b denote size of reading and writing from the buffer, respectively. As shown in the table, we can see that the I/O pattern of Montage shows high buffered I/O ratio, which is over 60% for all the datasets. Because we assume a buffer of adequate size, all written data can be buffered, i.e. $w_b = w_t$.

Large-scale HPC applications also have high temporal I/O locality because these applications usually write checkpoints for fault tolerance. In checkpoint/restart, applications read the *latest* checkpoint when restarting. The I/O pattern increases temporal I/O locality. Thus, improving read and write performance in r_b, w_b can accelerate these data intensive workloads.

2.3 Motivation of Burst Buffer in Clouds

As mentioned in Section 2.2, data intensive workloads with high temporal I/O locality stand to benefit from their I/O data being buffered. If we can install *large*, *fast*, and *shared* buffer spaces, we can accelerate these data intensive workloads. Motivated by this fact, we enhance the performance of the storage hierarchy of cloud environments though the use of a *burst buffer* technology. Burst buffer is a new tier in current storage hierarchy for bursty I/O operation in data intensive applications [9] and checkpointing workloads [20]. By incorporating the new tier of storage into clouds, the bursty I/O workloads can be absorbed without a need of higher bandwidth storage.

Furthermore, our investigation on cloud network shows

	data set 0	data set 1	data set 2
Input data set size (MB)	25	182	1200
Output data set size (MB)	76.5	574.3	7100
Total I/O size (MB)	224.1	2132.4	30250.018
Total read size(MB)	147.6	1558.1	22667.261
Total write size(MB)	76.5	574.3	7582.757049
Temporal I/O locality size(MB)	139.14	2074.5	29609.493
R_{buf}	62%	97.3%	97.882%
Single Thread Compute time (s)	4.7	18.7	311.82

Table 1 Montage Execution detail

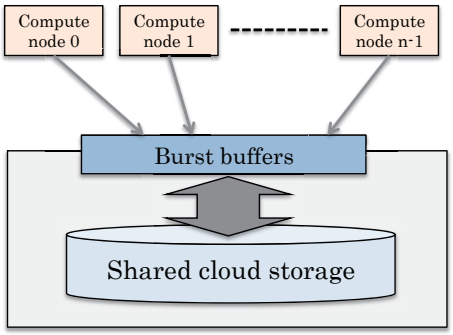

Fig. 5 Overview of cloud-based burst buffers

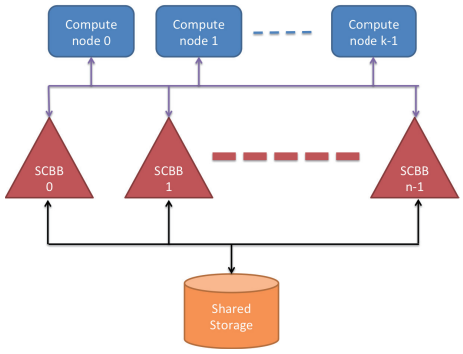

Fig. 6 Architecture of cloud-based burst buffers

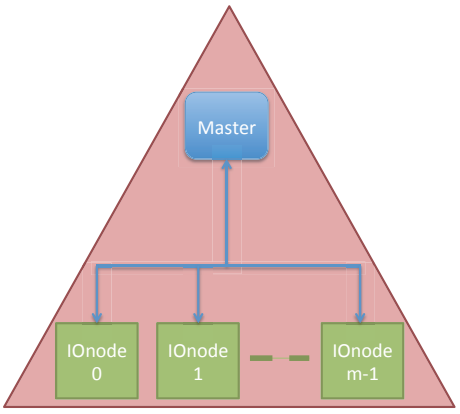

Fig. 7 Structure of SCBB

that the LAN networks inside clouds exhibit high and stable performance. Fig. 4 shows point-to-point communication throughput in Amazon EC2. In this evaluation, we measure the network throughput between a pair of *m3.xlarge* instances using Iperf [21], and increase the number of the pairs. The specification of the instances is in Table 3. As shown in the figure, we can see that the network throughput between two nodes is high and proportional to the increasing number of the pairs. Hence we take advantage of the high and scalable network throughput to burst I/O throughput. If we use the combined memory space from a group of instances as burst buffers, we can construct on-demand burst buffers with high throughput and high scalability on the fly.

3. Architecture of Our Proposed System

As we mentioned in Section 2, we have two major challenges when executing HPC data intensive applications on cloud: low and unstable I/O performance causes reduction of application performance and eventual consistency causes job failure. In order to solve these two problems, we propose our cloud-based burst buffer system (Section 3.1). We take advantage of high and stable internal point to point connection to build local-system burst buffer to accelerate and stabilize the I/O performance and propose hybrid Master/Client and Key/Value model to distribute the I/O workload to achieve high scalability (Section 3.2 and 3.3). Then we handle the eventual consistency issue by maintaining data consistency among burst buffer nodes.

3.1 Overview of Cloud-based Burst Buffers

In this section we introduce the overview of our cloud-based burst buffer system. Fig. 5 shows the role of our proposed system in cloud storage environment, In this figure, *Compute Nodes* refer to the nodes on which applications run. *Burst buffers* are provided by the same instances as *Compute Nodes*. Unlike *Compute Nodes*, the instances of the *burst buffers* provide memory buffers remotely for caching data. When the applications read or write data, the I/O operations are handled via *burst buffers*. The *shared cloud storage* is a persistent shared storage such as Amazon S3. The caching operation via burst buffers is agnostic to the applications, therefore, the applications can benefit from two-levels of storage hierarchy without knowing the underlying operations/architecture.

3.2 Our Proposal hybrid Master/Client and Key/Value Architecture

As mentioned in Section 2.1, I/O throughput and file

	Master/Client	Key/Value
pros:	Easy to monitor the entire system performance ,load balance	No central node, easy to scale out.
cons:	Master node can easily become the bottleneck of the whole system	difficult to monitor the whole system, and balance the work load

Table 2 Advantage and Disadvantage of Existing Models

metadata operations can both greatly affect the performance of data intensive applications. The performance can be further impacted by possible imbalance of workload and, therefore, it is imperative that the I/O performance of the entire system be monitored. However, the existing distributed file system models, like Master/Client and Key/Value, cannot simultaneously achieve high I/O performance as well as monitor and control the system as shown in Table 2. For this reason, we propose a hybrid model of Master/Client and Key/Value that combines the benefits of both models to overcome the individual limitation of each.

Next, we describe how we apply our proposal hybrid Master/Client and Key/Value in our burst buffer system. Fig. 6 and Fig. 7 show the overview architecture of our proposed burst buffer system. The system consists of several Sub Cloud-based Burst Buffers (SCBBs) and each SCBB consists of a single *Master Node* and several *IOnodes*. In each SCBB, the *Master Node* is in charge of controlling all *IOnodes* and managing file metadata, whereas *IOnodes* are responsible for storing the actual data and transferring data with *Compute Nodes*. The most common problem of using a Master/Client architecture is that the master can easily become the bottleneck of the whole system. In order to avoid the problem, we split the entire fileset into several sub filesets. Each SCBB is responsible for one of the sub filesets and the hash of file's path is used to decide to which SCBB each file belongs to. This allows us to distribute workload among several SCBBs and make our system more scalable. Inside each SCBB, *Master Node* can monitor the performance and balance the workload of *IOnodes*. Furthermore, since it is difficult to buffer all the I/O data in *IOnodes*, swap-out and write-back operations are necessary. *Master Node* monitors the system and controls these operations.

All *Master Nodes* should be set up at the beginning and remain unchanged. The information of all *Master Nodes* are stored in all *Compute Nodes*. Since all *Master Nodes* remain unchanged, *Compute Node* can use following formula to decide the corresponding *Master Node*:

$$master = masterList[hash(filePath) \mod N] \quad (2)$$

here *masterList* is a list of all *Master Nodes* in the system N is the number of *Master Nodes*.

3.3 Structure of SCBB

Here we introduce the structural details of each SCBB.

3.3.1 *Master Node*

Master Nodes are the supervisors of their SCBBs. *Master Nodes* store file metadata of each buffered file including file size, file access time etc. Each *Master Node* also maintains a list of the *IOnodes* in the its SCBB and a file-*IOnode* map, which shows the files that are buffered in each *IOnode*.

Besides controlling the SCBB and *IOnodes*, *Master Nodes* are also responsible for handling all the I/O operations from *Compute Nodes* such as file open, flush, close etc. Furthermore, in order to monitor and balance the workload, control swap-out/write-back, I/O operations like read and write go through *Master Nodes*. *Compute Node* doesn't cache file-*IOnodes* map and require the map information from *Master Nodes* before each read/write operations. This helps in maintaining file consistency among all the clients. *Master Nodes* are, however, not involved in the actual transfer of data. For data transfers, *Compute Nodes* query the file-*IOnodes* maps and then connect directly to the corresponding *IOnodes* directly for sending and receiving data. This serves to reduce the burden on *Master Nodes*.

3.3.2 *IOnodes*

IOnodes store the actual data under the control of its *Master Node* and are also responsible for the actual data transfer. In each SCBB there can be several *IOnodes*, which can be set up to mitigate the system stress and perform shutdown to reduce the monetary cost on demands.

4. Implementation

In order to validate our proposal in real cloud system, we implemented our cloud burst buffer system with FUSE framework. In this section, we show the overview of the implementation (Section 4.1), describe the optimizations for performance (Section 4.2), and details of various file operations on our system (Section 4.3).

4.1 Implementation Overview

As described in previous sections, there are three kinds of nodes in our system: *Master Nodes*, *IOnodes*, *Compute Nodes*. Our implementation is based on the TCP/IP protocol and use sockets to communicate among *Master Nodes*, *Compute Nodes*, and *IOnodes*. We implement *Master Node* and *IOnodes* as servers and *Compute Nodes* as clients. To avoid slow down caused by hard disks, we buffer files in the main memory of each *IOnode*.

In order to intercept all the I/O operations from applications, and let users access data as other local directories, we implement *Compute Nodes* with FUSE [22]. It provides several hooks to intercept I/O operations under mounted directories, such as open, read, write, flush etc. We implement these hooks and redirect these I/O operations to corresponding *Master Nodes* and *Compute Nodes*.

Although FUSE supports multi-threaded operations, we

only support single thread in the current implementation. As mentioned in Section 5, although our proposed architecture, *Master Nodes* can control and balance the work load in each SCBB, in this paper, we focus on the effectiveness of cloud-based burst buffer, so in current implementation, we use simple round-robin policy to assign file to *IOnodes* in each SCBB. In addition, we assume that all the files can be buffered in *IOnodes* and we don't perform write back. We also don't implement data replication for fault tolerance and data reliability in the current implementation. Although there are several ways to maintain the data consistency among burst buffer nodes when file data is replicated, currently the files are not replicated hence there is no eventual consistency problem described in Section 2.1 in our implementation. However these we mentioned above are the subjects of our future work to make our system more effective and robust.

4.2 Optimizations for Performance

FUSE makes it easy to intercept I/O operations and implement a userspace filesystem, nonetheless, targeted optimizations are required to achieve high performance. In this section, we introduce these optimizations in our implementation.

Firstly, we reduce I/O latency by reusing sockets. In TCP/IP protocol, creating a new socket needs three way handshakes and causes slow start. When each *IOnodes* starts up, it first registers itself to its *Master*, and when *Compute Nodes* start up they register themselves to all of the *Master Nodes*. They keep the sockets alive and reuse them for subsequent communications. *Compute Nodes* also register themselves to *IOnodes* at the first time they communicate to each other, not at the start up, so that *IOnodes* can be increased and decreased on demand. When *Compute Nodes* and *IOnodes* are removed, they unregister themselves from *Master Node* and *Compute Nodes* also unregister themselves from all the registered *IOnodes*.

Secondly, we split files into multiple fixed-size chunks to handle a large number of files and reduce unnecessary data transfer unlike s3fs, which requires a file to be read/written in its entirety. This optimization enables *IOnodes* to store only necessary chunks of files so they can increase the number of files they can buffer. In addition, this optimization also reduces unnecessary data transfer between s3 and compute nodes when only a small portion of file is read or updated. We adopt a lazy data allocation policy, which means we allocate data chunk and read data from remote shared storage only when necessary. Additionally, we introduce dirty flags for each data chunk, which works like dirty flags in conventional memory management techniques. The flag is set to CLEAN when a chunk is first read from the remote shared storage, signifying that the buffered chunk is up to date. When the data chunk is updated, the flag changes to DIRTY and only DIRTY chunks need to be written back during the write-back phase; this reduces the data transferred for write operations. Although distributing chunks of a single file over multiple *IOnodes* can achieve further performance and scalability, deciding the best chunk placement is difficult. In the current implementation, thus, we buffer all chunks of a single file are buffered in a single *IOnode*.

Thirdly, we introduce an additional tier of buffers, local I/O buffers, to reduce the number of read/write sub-requests in the FUSE framework. In FUSE version 2.9.3, it divides read/write requests into sub-requests and each sub-request handles one memory page, which is typically 4KB in common Unix systems. This increases the read/write latency and requests to a huge file lead to a huge number of sub-requests and *Master Nodes* possibly become bottleneck because *Compute Nodes* need to communicate with *Master Nodes* and *IOnodes* before actual data transfer on every request thus all the I/O operations go through *Master Nodes*, although we use hybrid Master/Client and Key/Value architecture to distribute workload and *Master Nodes* are not involved in actual data transfer. We introduce local I/O buffer, like buffered I/O in common Unix systems, such as fread and fwrite. A local I/O buffer is allocated for each opened file on *Compute Nodes* and read/write operations within the buffer range can complete locally in the *Compute Node*. We also use dirty flags to local I/O buffers like flags for file chunks. A local I/O buffer is written back to the *IOnode* only when the flag is DIRTY. Actual data transfer between *IOnodes* and *Compute Nodes* happen in the following situations:

- Read, write or seek out of local buffer buffer range.
- FUSE framework calls flush on file with DIRTY dirty flag.
- FUSE framework calls close on file with DIRTY dirty flag.

Finally, when using FUSE, besides read/write, other file operations are also called frequently, such as getattr, which means get the file status. Since all of the *Master Nodes*, *IOnodes* and *Compute Nodes* are not multithreaded in current implementation, it will causes high latency and slow down other file operations if all these file meta-data operations all go through *Master Nodes*. To prevent this circumstance, we create local buffers to cache file meta data to accelerate these file-meta operations. However, we cannot buffer all the file-meta data in order to keep consistency among all the *Compute Nodes*, because the cached meta data will be out of date when other nodes update that file. Hence in our implementation, only the meta data of opened files are buffered. When a user opens a file, *Compute Node* retrieves file-meta data from corresponding *Master Node*, and buffers it for subsequent file-meta operations, and when user application updates meta data, the operations will be performed on these meta data buffer and the buffer is marked as DIRTY. When flush operation is called for the file whose buffer is marked as DIRTY, *Compute Node* will first update *Master Nodes*'

meta data with locally buffered meta data, and then transfer data to *IOnodes*.

4.3 File Operations

In describe how we implement the system and optimizations in detail, and how our system handle each I/O operations. In this section, we give more details about the file operations in our implementation: open, create, flush, close, read, write and other operations in FUSE.

4.3.1 open and create

When an application opens a file, the FUSE framework calls our open function, and pass the relative file path from the mount point, and other information like file access mode. We first apply formula 2 against the file relative path to select the corresponding *Master Node*, and record this information in local data structure, then send the file path and file open flag to that *Master Node*, when *Master Node* receives the path, it first searches whether the file has been buffered already, If the file is already buffered, *Master Node* replies to the client with a unique id which will be used in the subsequent file operations. If the file hasn't been buffered, *Master Node* will first pick up a new file id, and select several *IOnodes* in the same SCBB, and send open request with the file relative path to these *IOnodes*, subsequently, *Master Node* returns the file id to *Compute Node*. *Compute Node* stores the file id in FUSE file meta-data and uses it for the following operations.

File creation is quite similar with file open, when *Compute Node* connects to *Master Node*, beside the file relative path and open flag, we also send the file mode, which denotes the permission in common Unix system. In order to accelerate file creation, *Master Node* creates a new file with the file mode, assigns some *IOnodes*, and return the file id. In our implementation *Master Node* don't actually create a new file in remote shared storage, rather, *Master Node* marks the new created file with a not existing flag, and create file in remote shared storage when write back.

4.3.2 read and write

When an application issues a read request, the FUSE framework calls our read function with file id, data buffer pointer, read size and offset. We first get the corresponding *Master Node* from local record, and then send the read request with read size and offset. As mentioned in section 4.2, files are split into several fixed-size data chunks, after receiving these information, *Master Node* responds to the *Compute Node* with a map of file chunk-*Onodes*, provides *Compute Node* with the information about which *IOnode* is responsible for each file chunk. After receiving the map, *Compute Node* connect to the corresponding *IOnode* directly, and sends the read request with size and offset. If the file is buffered in the *IOnode*, the *IOnode* will send *Compute Node* with the requested data, otherwise, the *IOnode* will first read from remote shared storage, can then send the data.

Write operation is treated similarly to the read operation. In addition to the read case, we also have to consider the case that data is written beyond the current end of file. Since files are split into several fixed-size data chunks, data chunk needs to be additionally allocated in this case. When *Master Node* finds that data chunk allocation is required to fulfill write request by a *Compute Node*, *Master Node* will send the allocation request to the same *IOnode* which buffers the existing data chunks of the same file, as one file will be buffered in only one *IOnode* in current implementation. Subsequently *Master Node* appends this new chunk-*IOnode* pair into its chunk-*IOnode* map, and responds to the *Compute Node* with the new chunk-*IOnode* map.

4.3.3 flush and close

As mentioned in section 4.2, in order to reduce the read/write operations between *Compute Nodes* and *masters*, *IOnodes* as well as the unnecessary write back data transfer, we introduce the local I/O buffer, and dirty flag. Hence when flush is called, *Compute Node* first checks the dirty flag of local I/O buffer. If it is set to DIRTY, *Compute Nodes* will call write on local I/O buffer, and set the dirty flag to CLEAN.

When user closes the file, the flush operation will be invoked by FUSE framework first, and then the close request will be sent to *Master Node* to perform actual file close.

4.3.4 other file operations

In FUSE framework, other than the abovementioned operations, there are also several file meta data operations available such as getattr, which means get file status. We also implemented this and other file meta data operations in order to make our system behave correctly.

5. Evaluation

To validate the effectiveness of cloud-based burst buffers, we conduct several evaluations in the Amazon EC2 public cloud. For the Clients, IOnodes and Masters we use Amazon EC2 instances shown in Table 3. All the Clients, IOnodes and Masters node connects with interconnection network inside Amazon EC2 in the same region.

System	Amazon EC2
Region	Tokyo
Instance Type	m3.xlarge
vCPUs	4
ECUs	13
Memory	15GiB
Instance Storage	2*40GB(SSD)
Network Performance	High
Cost	$0.405 per Hour
AMI Type	Amazon Linux AMI (HVM)

Table 3 Evaluation Environment

Here, vCPUs means the number of virtual CPUs in the instance, and a single ECU (Amazon EC2 Compute Unit) provides the equivalent CPU capacity of a 1.0-1.2GHz 2007 Opteron or 2007 Xeon processor [1]. For the shared cloud storage, we use Amazon S3 and mount it onto all IOnodes

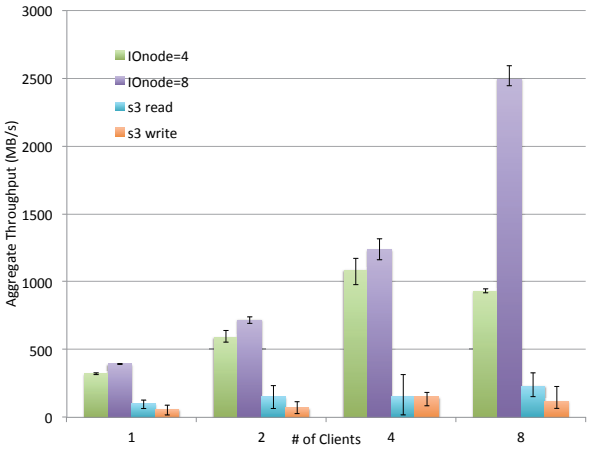

Fig. 8 Sequential I/O Performance

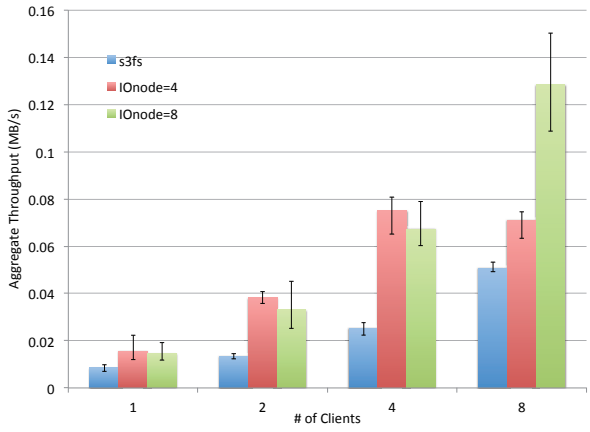

Fig. 9 Random I/O Performance

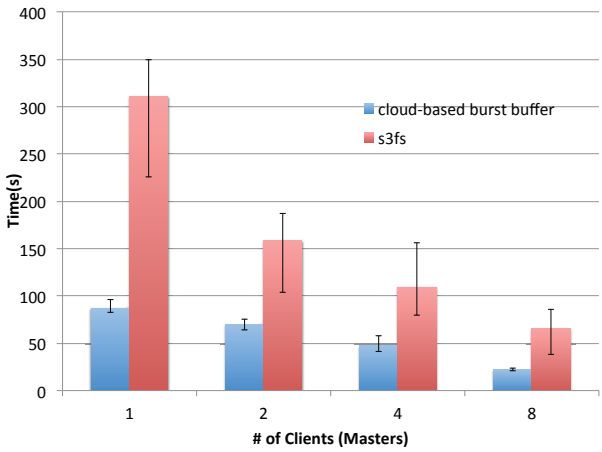

Fig. 10 File Creation Performance

and Masters using s3fs [23]. We set the data chunk size in *IOnodes* and local I/O buffer size to 100MB.

5.1 I/O Performance of Cloud-based Burst Buffer

First, To illustrate how our system solves the performance issue we mentioned in Section 2, we show how much our system can improve and stabilize I/O performance with different I/O patterns. We evaluate the performance of two basic I/O patterns: sequential and random I/O, on our system. Since the applications' I/O patterns are the combination of these two basic patterns, if we can improve the performance of these two basic patterns, we can improve the applications' I/O performance. We measure the sequential and random I/O performance with different number of *IOnodes* in a single SCBB. Data is sent from burst buffers to compute nodes when we read data from burst buffers, and vice versa when we write. Thus, both performances are identical and shown in the single bars in the Fig. 8 and 9. Each client accesses data of size 100MB in sequential access case and accesses one byte for 10,000 times from random offsets of a 100MB file in random access case. In order to achieve high random access, both s3fs and our system provide local I/O buffers.

The sequential I/O performance is shown in the Fig. 8, we can see that the performance of our system increases as the number of IOnode/Client pair increases. When the number of clients is smaller than the number of IOnodes, the performance scales out as the number of clients increases, however, when the number of clients is larger than the number of IOnodes, the performance cannot improve any further. This is because the maximum I/O throughput is limited by the total bandwidth of IOnodes. With 8 clients and 8 IOnodes, we can achieve the aggregated throughput up to 2,500 MB/s. On the other hand, without our cloud-based burst buffers, we see that applications can only achieve 330 MB/s for read and 250 MB/s for write using 8 nodes. Next, we show the random I/O performance in Fig. 9. Thank to local buffer, we can see that both our system and s3fs performances scale well and get stable performance. We get similar results as sequential I/O cases. Our system again provides higher performance than s3fs.

From these two experiments we show that our system scales significantly as the number of IOnode/Client pair increases and can remarkably increase the two basic I/O performances, sequential and random I/O compared to s3fs.

5.2 Scalability of SCBB

As we mentioned in Section 2, besides the I/O throughput, other meta data operations can become bottleneck of I/O performance. Furthermore since the main drawback of Master/Client model is scalability, because Master can easily become bottleneck of entire system, hence we propose our hybrid model to alleviate such bottleneck in Section 3. In order to validate the effectiveness of our hybrid model, we evaluate the file creation performance to show the scalability of our system. We chose file creation performance since it is typical meta-data operation and has been widely used to evaluate meta-data performance and scalability of filesystems. We evaluate the file creation performance of creating 1,000 files totally with different number of SCBBs. Since *Master Nodes* and *Compute Nodes* both operate in single thread,

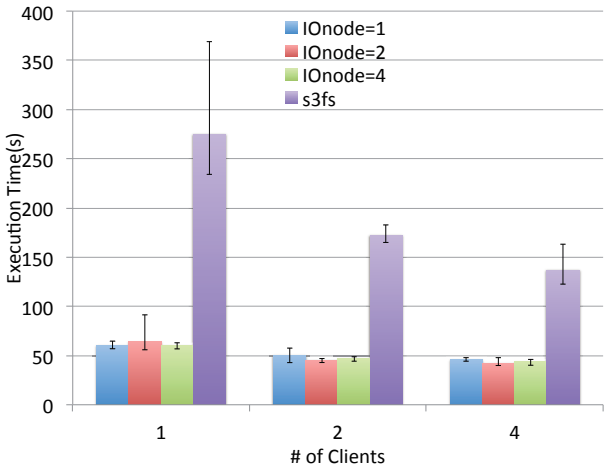

Fig. 11 Execution time of Montage on Multiple Nodes

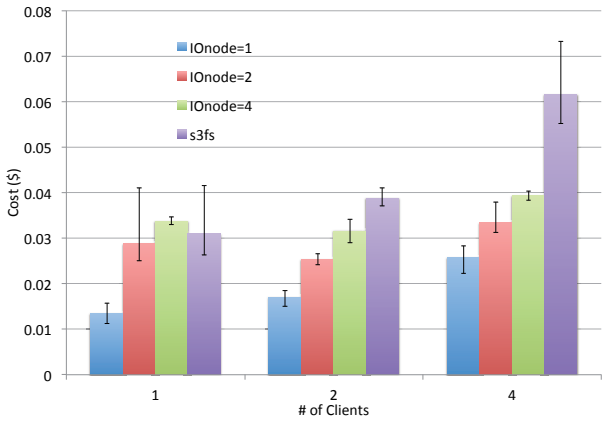

Fig. 12 Cost for Execution of Montage

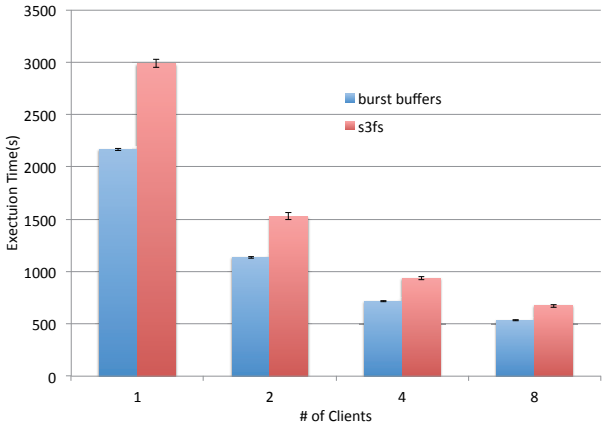

Fig. 13 Execution time of Supernovae on Multiple Nodes

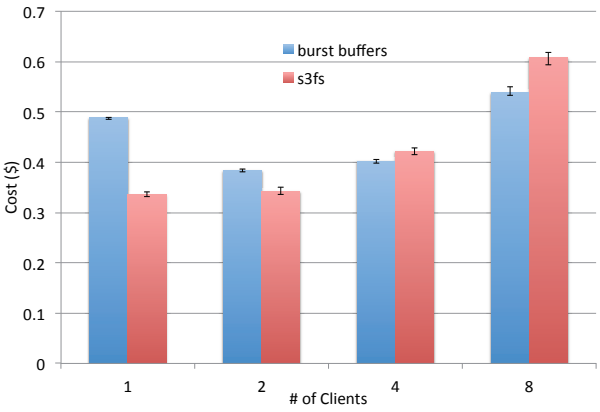

Fig. 14 Cost for Execution of Supernovae

in this experiment, we set the number of *Compute Nodes* equals to the number of *Master Nodes*. Since we use hash of file path to select corresponding *Master Node*, we randomly change the file path for each measurement.

As we can see from Fig. 10, by increasing the number of *Master Nodes*, our system can scale out significantly. Since we use hash of a random file path to select a *Master Node*, achieving a perfectly balanced workload every time is difficult, but we can expect a statistically balanced workload when the number of files increases. We can achieve 2.5 times performance improvement by using our system compared to S3. From this experiment we show that our system can greatly improve the meta data performance compared to S3 and our proposal hybrid model can alleviate the bottleneck caused by common Master/Client model.

5.3 Accelerate Real Application

In previous sections, we present how our system can accelerate all basic I/O patterns as well as meta data operations compared to Amazon S3. In order to illustrate how such improvement in I/O performance affects real applications, we evaluate two real world HPC applications on our system in this section, Montage and Supernovae. In this experiment, we use GXP [24] to distribute workflow on multiple nodes and compare the performance between our system and s3fs. Since Amazon S3 adopts eventual consistency as we mentioned previously, we observed frequent job failures due to the inconsistent read, hence in this experiment we slow down the execution by disabling the local file status cache in s3fs to ensure consistent read. On the other hand, there is no consistency issue when using our system. In Montage experiments we evaluate the performance using different number of IOnodes to present how different number of IOnodes affects data intensive applications' performance. However in Supernovae cases, since supernovae doesn't have as heavy as I/O burden like Montage, we set the Master and only one IOnode process on the same node to reduce the cost. The data set we used is data set 0 in Table 1.

First we show the Montage results. Fig. 11 shows the results of the execution of Montage. As we can see, our system can improve the performance greatly (over 4.58 times on the single node cases). In multiple node cases, the performance improved by increasing the clients seems comparatively small. This is because the workflow applications like Montage have dependencies in process (Fig. 1), and in our test case, it cannot run perfectly parallel. For the same reason, increasing the number of *IOnodes* can provide a limited performance improvement.

Although the data set used in our experiment is small, as we discussed in Section 2.2, further improvement can be expected when using larger data set such as data size 1 or 2 in Table 1. Next, Fig. 13 shows the results of Supernovae. As we can see, we get similar results as Montage cases, we can accelerate the execution by using our system. Moreover we can see that the performance improves greatly as the number of clients increases, it is because Supernovae also has heavy computation, hence the performance improves as the computational resources increased. We also observed the performances of both s3fs and our system are more stable compared to Montage cases, since Supernovae has less I/O than Montage, hence the performance bears less influence from the I/O performance fluctuation.

In addition, we found that although we use additional nodes in our system, the overall monetary cost can still improved due to the reduction of execution time. Fig. 12 and 14 shows the comparison of monetary cost for the Montage and Supernovae execution respectively. As shown in the figures, in both cases we observed cost reduction by using our system. Moreover, in Montage, the data intensive application cases, due to the huge reduction in execution time, we can achieve up to 56.3% reduction of the monetary cost in addition to the I/O performance. Although, the environments with cloud-based burst buffers use more Amazon EC2 instances, i.e. additional IOnodes and Master Node, the execution is cost effective because the execution time can be greatly reduced by using our cloud-based burst buffers.

In some cases, the cost increases because of low contribution of increasing *IOnodes*. In such cases, if we can dynamically change the number of burst buffers during execution according to I/O time and the computation time, we can increase the I/O throughput while minimizing the monetary cost. In addition, if we dynamically change I/O destinations between cloud-based burst buffers and S3, we can optimize both I/O throughput and the monetary cost depending on the types of applications. We can analyze the I/O patterns based on MUSE and then use the results to optimize the workloads. We will consider the I/O tuning in future work.

6. Related Work

Burst buffer has been used in several research as a new design in storage system to fill the performance gap between node's local storage throughput and parallel file system throughput. Liu et al. [9] proposed burst buffers for high performance systems to absorb bursty I/O request from applications. They analyzed the I/O patterns of several applications and the workload of the whole system with their proposed burst buffer and delivered its effectiveness. Sato et al. [25] explored effectiveness of burst buffers in checkpointing and proposed checkpoint strategy for burst buffers.

The emergence of a cloud computing technology brings a new solution to large scale high performance computing with low initial costs, high accessibility, and flexibility. There are already several research using public clouds for large scale computation. Garcia et al. [26] deployed *Hadoop* on Amazon EC2 for querying large web public datasets. Wittek et al. [27] combined an implementation-independent workflow designer with cloud computing to support small institution in ad-hoc peak computing needs. Anthony et al. [28] discussed the applicability of cloud computing to large-scale satellite ground systems. However, these work only focus on how to run these large scale application on clouds, and do not consider the problem of increased I/O throughput to shared cloud storage.

I/O throughput problem is a critical part in cloud computing and many works have been focusing on this problem. Hovestadt et al. [29] considered another way to improve the I/O performance, they proposed a new adaptive compression scheme for virtualized environments and improved the I/O performance by compressing the I/O data. Hongtao Du et al. [30] focused on the performance of meta-data server in cloud environments. They proposed a high throughput system for cloud storage by building meta-data server on the high performance mainframe. Meanwhile, we extend and apply the state-of-the-art burst buffer technology for clouds to accelerate I/O performance. To the best our knowledge, this work is the first explorations of innovating burst buffer technologies in order to solve low I/O throughput problems in clouds.

7. Conclusion and Future Work

A cloud-based burst buffer system has been introduced in order to accelerate I/O performance of data intensive HPC applications running in the cloud. We implemented our proposal using a hybrid of Master/Client and Key/Value models, capable of simultaneously achieving both scalability and good load balancing.

Our benchmarks have shown that our system accelerates and stabilizes read throughput, write throughput, and meta data operations by up to 10.47, 16.8, and 2.5 times, respectively, on Amazon EC2/S3. Moreover, our system can perfectly solve the eventual consistency issue in Amazon S3, which causes frequent job failures. The executions of Montage and Supernovae show that we can improve performance of a real world data intensive application by up to 4.5 times as well as save 56.3% monetary cost.

As future work, we will extend the system to be able to dynamically change the number of burst buffer nodes depending on the I/O workload, thereby further increasing I/O throughput and minimizing costs.

Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-673830). This research was supported by

JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration). This research made use of Montage, funded by the National Aeronautics and Space Administration's Earth Science Technology Office, Computation Technologies Project, under Cooperative Agreement Number NCC5-626 between NASA and the California Institute of Technology. Montage is maintained by the NASA/IPAC Infrared Science Archive.

References

[1] : Amazon AWS, http://aws.amazon.com/.
[2] : Azure, http://azure.microsoft.com/.
[3] : IBM Public Cloud, http://www.ibm.com/cloud-computing/us/en/.
[4] : Oracle Cloud, https://cloud.oracle.com/home.
[5] : HP Helion Public Cloud, http://www.hpcloud.com/.
[6] Amazon: Amazon AWS HPC Instance, http://aws.amazon.com/hpc/.
[7] Bermbach, D. and Tai, S.: Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3's Consistency Behavior, *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, New York, NY, USA, ACM, pp. 1:1–1:6 (online), DOI: 10.1145/2093185.2093186 (2011).
[8] Golab, W., Rahman, M. R., Young, A. A., Keeton, K., Wylie, J. J. and Gupta, I.: Client-centric Benchmarking of Eventual Consistency for Cloud Storage Systems, *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, ACM, pp. 28:1–28:2 (online), DOI: 10.1145/2523616.2525935 (2013).
[9] Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A. and Maltzahn, C.: On the role of burst buffers in leadership-class storage systems, *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pp. 1–11 (online), DOI: 10.1109/MSST.2012.6232369 (2012).
[10] : Montage, http://montage.ipac.caltech.edu/docs/grid.html.
[11] : Amazon simpleDB, http://aws.amazon.com/simpledb/.
[12] rklundt: HP Helion Object Storage Service, http://docs.hpcloud.com/publiccloud/api/object-storage/.
[13] : Rackspace Cloud Files, http://www.rackspace.com/blog/storage-systems-overview/.
[14] Chiba, T., den Burger, M., Kielmann, T. and Matsuoka, S.: Dynamic Load-Balanced Multicast for Data-Intensive Applications on Clouds, *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, Washington, DC, USA, IEEE Computer Society, pp. 5–14 (online), DOI: 10.1109/CCGRID.2010.63 (2010).
[15] Gropengiesser, F. and Sattler, K.-U.: Transactions a la carte - Implementation and Performance Evaluation of Transactional Support on Top of Amazon S3, *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1082–1091 (online), DOI: 10.1109/IPDPS.2011.252 (2011).
[16] Yoon, H., Gavrilovska, A., Schwan, K. and Donahue, J.: Interactive Use of Cloud Services: Amazon SQS and S3, *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pp. 523–530 (online), DOI: 10.1109/CCGrid.2012.85 (2012).
[17] Palankar, M. R., Iamnitchi, A., Ripeanu, M. and Garfinkel, S.: Amazon S3 for Science Grids: A Viable Solution?, *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, DADC '08, New York, NY, USA, ACM, pp. 55–64 (online), DOI: 10.1145/1383519.1383526 (2008).
[18] Garfinkel, S. L. and Garfinkel, S. L.: An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS, Technical report, Center for (2007).
[19] : MUSE, https://github.com/kento/MUSE.
[20] Sato, K., Mohror, K., Moody, A., Gamblin, T., de Supinski, B., Maruyama, N. and Matsuoka, S.: A User-Level InfiniBand-Based File System and Checkpoint Strategy for Burst Buffers, *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pp. 21–30 (online), DOI: 10.1109/CCGrid.2014.24 (2014).
[21] : Iperf, http://iperf.fr/.
[22] : FUSE, http://fuse.sourceforge.net/.
[23] : s3fs, https://code.google.com/p/s3fs.
[24] : GXP, http://www.logos.ic.i.u-tokyo.ac.jp/gxp/index.php?FrontPage.
[25] Sato, K., Maruyama, N., Mohror, K., Moody, A., Gamblin, T., de Supinski, B. R. and Matsuoka, S.: Design and Modeling of a Non-blocking Checkpointing System, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 19:1–19:10 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2388996.2389022⟩ (2012).
[26] Garcia, T. and Wang, T.: Analysis of Big Data Technologies and Method - Query Large Web Public RDF Datasets on Amazon Cloud Using Hadoop and Open Source Parsers, *Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on*, pp. 244–251 (online), DOI: 10.1109/ICSC.2013.49 (2013).
[27] Wittek, P., Jacquin, T., Dejean, H., Chanod, J.-P. and Daranyi, S.: XML Processing in the Cloud: Large-Scale Digital Preservation in Small Institutions, *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1072–1081 (online), DOI: 10.1109/IPDPS.2011.253 (2011).
[28] Anthony, R., Fritz, J. and Barnhart, D.: Cloud computing applications for large-scale satellite ground systems, *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, pp. 1894–1898 (online), DOI: 10.1109/MILCOM.2011.6127590 (2011).
[29] Hovestadt, M., Kao, O., Kliem, A. and Warneke, D.: Evaluating Adaptive Compression to Mitigate the Effects of Shared I/O in Clouds, *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1042–1051 (online), DOI: 10.1109/IPDPS.2011.256 (2011).
[30] Du, H. and Li, Z.: DHFS: A High-Throughput Heterogeneous File System Based on Mainframe for Cloud Storage, *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pp. 24–28 (online), DOI: 10.1109/PAAP.2011.13 (2011).