StorkCloud: Data Transfer Scheduling and Optimization as a Service

Tevfik Kosar, Engin Arslan, Brandon Ross, and Bing Zhang
Department of Computer Science & Engineering
University at Buffalo (SUNY)
Buffalo,New York 14260 USA
{tkosar, enginars, bwross, bingzhan}@buffalo.edu

ABSTRACT

Wide-area transfer of large data sets is still a big challenge despite the deployment of high-bandwidth networks with speeds reaching 100 Gbps. Most users fail to obtain even a fraction of theoretical speeds promised by these networks. Effective usage of the available network capacity has become increasingly important for wide-area data movement. We have developed a *"data transfer scheduling and optimization system as a Cloud-hosted service"*, StorkCloud, which will mitigate the large-scale end-to-end data movement bottleneck by efficiently utilizing underlying networks and effectively scheduling and optimizing data transfers. In this paper, we present the initial design and prototype implementation of StorkCloud, and show its effectiveness in large dataset transfers across geographically distant storage sites, data centers, and collaborating institutions.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*

General Terms

Design, Algorithms, Performance

Keywords

Big data; software as a service (SaaS); Cloud computing; end-to-end throughput optimization; data scheduling.

1. INTRODUCTION

As big-data processing and analysis dominates the usage of Cloud systems today, the need for Cloud-hosted data scheduling and optimization services increases. According to a recent study by Forrester Research [31], 77% of the 106 large IT organizations operate three or more datacenters and run regular backup and replication services among these sites. More than half of these organizations have over a petabyte of data in their primary datacenter and expect their inter-datacenter throughput requirements to double or triple over the next couple of years [27]. There has been a very similar trend in scientific applications for the last decade. Large scientific experiments such as environmental and coastal hazard prediction [23], climate modeling [21], genome mapping [7], and high-energy physics simulations [15, 4] generate data volumes reaching petabytes per year. Data collected from remote sensors and satellites, dynamic data-driven applications, and digital libraries and preservations are also producing extremely large datasets. This "data deluge" necessitates collaboration and sharing among the national and international research and industrial organizations, which results in frequent large-scale data movement across widely distributed sites.

Several national and regional optical networking initiatives such as Internet2, ESnet, XSEDE, and LONI provide high-speed network connectivity to their users. Recent developments in networking technology provide scientists with high-speed optical links reaching 100 Gbps in capacity [1]. Regardless, the majority of users fail to obtain even a fraction of the speeds promised by these networks due to issues such as sub-optimal protocol tuning, inefficient end-to-end routing, disk bottlenecks, and processing limitations. For example, Garfienkel reports that sending a 1 TB forensics dataset from Boston to the Amazon S3 storage system took several weeks [17]. For this reason, many companies prefer sending their data through a shipment service provider such as UPS or FedEx rather than using the Internet [14].

We have developed a Cloud-hosted data transfer and optimization service, StorkCloud, which will mitigate the large-scale end-to-end data movement bottleneck by efficiently scheduling and optimizing data transfer tasks, and effectively utilizing underlying networks. This Cloud-hosted data transfer and optimization service is based on our experience with Stork Data Scheduler [26, 25].

StorkCloud provides a Software-as-a-Service (SaaS) approach to the planning, scheduling, monitoring, and management of data placement tasks over wide-area networks. It implements application-level models, algorithms, and tools to predict the best combination of protocol and end-system parameters for end-to-end data-flow optimization of wide-area transfers. Optimizations include the number of parallel data streams; control channel pipelining and transfer concurrency levels; as well as integration of disk and CPU speed parameters into the performance model to predict the optimal number of disk (data striping) and CPU (parallelism) combinations for the best end-to-end performance. In this paper, we present the initial design and prototype implementation of StorkCloud.

2. STORKCLOUD DESIGN

The major components of StorkCloud include a multi-protocol transfer scheduler with modular throughput optimizers for queuing, scheduling, and optimizing data transfers; a transfer monitoring service (TMS) for providing the clients with real-time transfer updates; a directory listing service (DLS) for prefetching and caching remote directory metadata in the Cloud to minimize response time to the users; and a pluggable protocol interface which can communicate and negotiate with different data transfer protocols and storage systems. StorkCloud schedules, optimizes, and monitors data transfer requests from users through its thin clients (such as smartphone and tablet apps and a Web interface). It exposes APIs through an HTTP Representational State Transfer (REST) interface that clients may utilize to interact with services provided by StorkCloud. This makes the development of thin clients for StorkCloud very simple. Figure 1 presents an overview of the major StorkCloud components and their interaction.

2.1 StorkCloud Scheduler

StorkCloud's scheduler is a modular, multi-protocol task scheduler which handles queuing and execution of data transfer jobs and ensures their timely completion. The scheduler's plug-in interface allows arbitrary protocol support to be implemented as standalone modules and easily introduced to the system. The scheduler inherits much of its functionality from the Stork Data Scheduler [25]. As the core component of the StorkCloud system, its job is to take in transfer requests and provide information about transfer progress.

All communication between StorkCloud components is done with ClassAds [2] – a textual data representation format that is easy to parse and generate. The use of a text-based format allows for a standard communications interface that is accessible in any programming language capable of text manipulation. Serialized ClassAds are human-readable, allowing for easy debugging of communications code and interfacing with the scheduler using common utilities such as netcat or Telnet.

This led to a straightforward way of communicating between the scheduler and its plug-ins, which is necessary for exchanging information such as job progress. The information passed to the scheduler can be displayed to users or services interested in the progress of a job. This feature is used, for example, by the StorkCloud thin client interfaces to display job progress visually.

StorkCloud's scheduler can perform protocol-agnostic optimization of data transfers. Optimization modules, like transfer modules, can be plugged into the server, and incoming jobs can request an optimizer be used for the transfer. Optimization modules advertise which parameters they optimize, and transfer modules can either accept or reject the optimizer accordingly. If a transfer module allows an optimization to be used, it queries the optimizer for sample parameters, runs a sample, and reports the throughput back to the optimizer. The optimizer uses the reported information to determine parameters for the next sampling, and continues until either the transfer is complete or it is done sampling. This design allows optimizers to be protocol-agnostic; as long as the transfer module supports the features the optimizer exposes, neither needs to know the other's implementation details.

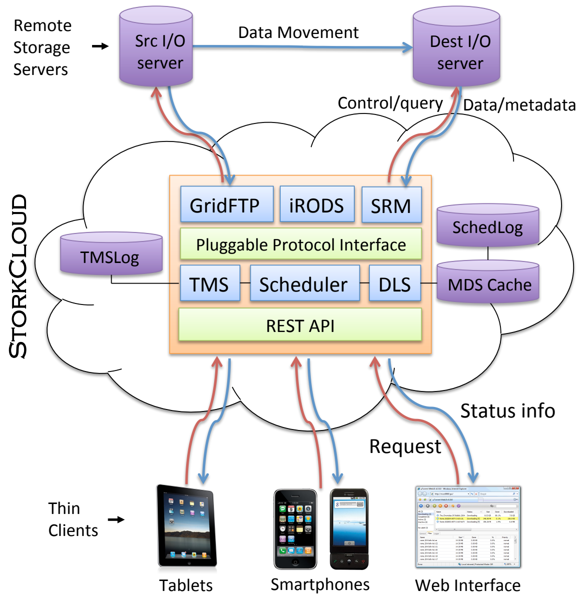

Figure 1: StorkCloud system components

2.2 Transfer Monitoring Service (TMS)

StorkCloud's Transfer Monitoring Service (TMS) provides clients with real-time monitoring information on the status of the data transfer jobs submitted. StorkCloud provides library utilities to help modules with bookkeeping and reporting transfer performance. Transfer modules are able to use these tools to generate useful and interesting progress information by simply reporting whenever files and blocks have been transferred. The timing and calculations necessary to generate progress information are performed by the library using this data, allowing the transfer modules to report job progress in a uniform way and freeing module implementors from having to track progress themselves. We are planning to extend this utility further in the future to produce other information as well – for example, estimated time until transfer completion.

StorkCloud will also provide additional reliability to Cloud data transfers via data transfer checkpointing, checksumming, and alternative protocol fallback mechanisms, which will be especially useful in large file transfers. We are developing an error reporting framework to distinguish the locus of failure (e.g, network, server, client, software, hardware), classify problems as transient or permanent, and provide recovery possibilities. These error detection, classification, and recovery mechanisms will provide greater reliability and agility to transfers performed by StorkCloud.

2.3 Directory Listing Service (DLS)

StorkCloud's Directory Listing Service (DLS) provides a multi-protocol file listing and metadata access service via a REST interface. Conceptually, DLS is an intermediate layer between clients and the remote servers which performs metadata operations on the client's behalf. In that sense, DLS acts as a centralized metadata server hosted in the Cloud. When a client wants to list a directory or access file metadata on a remote server, it sends a request containing necessary information (i.e., URL of the directory to list, along with credentials) to DLS, and DLS responds with the requested data.

During this process, DLS first checks if the requested metadata is available in its cache. If it is available in the cache (and the provided credentials match the associated cached credentials), DLS sends the cached information to the client without remotely connecting. Otherwise, it connects to the remote server, retrieves the requested metadata, and sends it to the client. Meanwhile, several levels of subdirectories will be prefetched at the background. Any cached metadata information will be periodically checked with the remote server to ensure freshness of the information. Clients also have the option to refresh/update the DLS cache on demand to make sure they are accessing the server directly, bypassing the cache. In future work, DLS's caching mechanism will be integrated with several optimization techniques in order to improve cache consistency and access performance.

Using thin client interfaces as a frontend for DLS, users can visually browse two remote listings simultaneously and initiate transfers with the press of a button.

2.4 Pluggable Protocol Interface

StorkCloud acts as a broker between clients and remote storage systems and protocols. In addition to the protocols StorkCloud supports out of the box, the Pluggable Protocol Interface allows users to develop custom modules to support different systems easily. Modules can be coded in any language recognized by the operating system, however users who want to have tighter integration with the system and better communication performance may implement transfer modules in Java to communicate directly with the StorkCloud scheduler in memory.

We are implementing a mechanism to enable inter-protocol translations to support data access from the Cloud to a wide variety of storage systems and data transfer protocols. To provide interoperability between different protocols, Cloud storage will be used as a temporary parking place for the data, while translation between the protocols is performed by StorkCloud. In the future, we may also offer direct access to file data through our HTTP interface, allowing other applications to support multiple transfer protocols using our service.

2.5 Throughput Optimization

StorkCloud also features a number of optimization algorithms implemented as optimization modules. We developed three highly-accurate models [38, 37, 22] which require as few as three sampling points to provide accurate predictions for the optimal parallel stream number. These models are demonstrably more accurate than existing models [19, 29] at predicting parallelism levels that maximize throughput. We have developed algorithms to determine the best sampling size and the best sampling points for data transfers by using bandwidth, Round-Trip Time (RTT), or Bandwidth-Delay Product (BDP) [33]. Our algorithms provide optimal throughput for wide-area links up to 10 Gbps bandwidth [34]. Currently, we are testing and tuning these models for higher bandwidth networks (40–100 Gbps) as well.

Tuning protocol parameters such as pipelining, parallelism, and concurrency levels significantly affects achievable network throughput. However, setting the optimal numbers for these parameters is a challenging problem, since poorly-tuned parameters can cause either under- or overutilization the network.

Among these parameters, **pipelining** specifically targets the problem of transferring a large numbers of small files. In most TCP-based transfers, the entire transfer must be acknowledged before starting the next transfer. This may cause a delay of more than one RTT between individual transfers. With pipelining, multiple unacknowledged transfers can be active in the network pipe at any time and the delay between individual transfers is minimized. **Parallelism** sends different chunks of the same file over parallel data channels (TCP streams) and achieves high throughput by aggregating multiple streams and getting an unfair share of the available bandwidth. **Concurrency** refers to sending multiple files simultaneously through the network using different data channels at the same time.

The models developed in our previous work lay the foundations of the StorkCloud optimizers presented in this paper, where we combine parallel stream optimization with pipelining and concurrency to achieve best possible network throughput. Different network channels can be used to transfer different portions (chunks) of the same dataset with different parameter combinations.

3. STORKCLOUD OPTIMIZERS

StorkCloud does not only provide data transfer as a service but also provides heuristic algorithms to optimize the data transfer throughput, called "optimizers". We designed StorkCloud in a modular way such that different optimizers may be used/specified for different needs. For example, if a user has access to a dedicated network, then fairness is not an issue as long as the transfer throughput improves. On the other hand, saturating the network and end-system resources for a single user's transfers is not considered to be fair and may even violate the resource utilization policies in shared networks.

In this paper, we present two StorkCloud optimizers: (i) the "Single-Chunk Concurrency (SCC)" approach which divides the set of files into chunks based on file size, and then transfers each chunk with its optimal parameters; and (ii) the "Multi-Chunk Concurrency (MCC)" approach which likewise creates chunks based on the file size, but rather than scheduling each chunk separately, it co-schedules and runs small-file chunks and large-file chunks together in order to balance and minimize the effect of poor performance of small file transfers.

3.1 Single-Chunk Concurrency (SCC)

Files with different sizes need different set of parameters for optimized transfers. As an example, pipelining and data channel caching would help mostly to small file transfers, whereas parallel streams would be beneficial if the files are large enough. Optimal concurrency levels for different file sizes would be different as well.

To analyze the effects of different parameters on the transfer of different file sizes, we conducted experiments for each parameters separately, as shown in Figure 2. We created a mixed dataset consisting of 10,000 x 1MB files, 100 x 100MB files and 10 x 10GB files for the XSEDE network; and another mixed dataset consisting of 10,000 x 1MB files, 100 x 100MB files and 10 x 1GB files for the LONI network. A smaller data set is sued for the LONI network due to disk storage limitations on LONI. We initially transferred each dataset by only changing one parameter (i.e. pipelining, parallelism or concurrency) at a time to observe the individual effect of each parameter.

Algorithm 1 Partitioning files into chunks based on file size

```
Require: List of all files and Bandwidth Delay Product
 1: function PARTITIONFILES(allFiles,BDP)
 2:     Partition Small, Middle, Large, Huge
 3:     while $allFiles.size() > 0$ do
 4:         $File f = allFiles.pop()$
 5:         if $f.size < \frac{BDP}{10}$ then
 6:             Small.add(f)
 7:         else if $f.size < \frac{BDP}{2}$ then
 8:             Middle.add(f)
 9:         else if $f.size < BDP * 20$ then
10:             Large.add(f)
11:         else
12:             Huge.add(f)
13:         end if
14:     end while
15:     PartitionList p;
16:     p.add(Small),p.add(Middle),p.add(Large),p.add(Huge)
17:     mergePartitions(p)
18: return Small,Middle,Large,Huge
19: end function
Require: Partition p, BDP, buffer size and maximum allowed concurrency
20: function FINDOPTIMALPARAMETERS(p)
21:     pipelining,parallelism,concurrecy
22:     Density d = findDensityofPartition(p)
23:     $avgFileSize = findAverage(p)$
24:     if $d == SMALL$ then
25:         $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil - 1$
26:         $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 1$
27:         $concurrency = Min(\frac{BDP}{avgFileSize}, p.count(), maxConcurrency)$
28:     else if $d == MIDDLE$ then
29:         $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil$
30:         $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 1$
31:         $concurrency = Min(\frac{BDP}{avgFileSize}, p.count(), maxConcurrency)$
32:     else if $d == LARGE$ then
33:         $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil + 1$    ▷ This chunk should have pipelining
34:         $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 2$
35:         $concurrency = Min(2, p.count(), maxConcurrency)$
36:     else if $d == HUGE$ then
37:         $pipelining = \left\lceil \frac{BDP}{avgFileSize} \right\rceil - 1$ ▷ Pipelining will be zero in most cases
38:         $parallelism = Min(\left\lceil \frac{BDP}{bufferSize} \right\rceil, \left\lceil \frac{avgFileSize}{bufferSize} \right\rceil) + 2$
39:         $concurrency = Min(2, p.count(), maxConcurrency)$
40:     end if
        return pipelining,parallelism,concurrency
41: end function
```

Although LONI and XSEDE network results in Figure 2 are little different, there is no doubt that concurrency is the most dominant parameter for all file sizes. This is the reason why we present our experiment results in section 4.1 on "Performance Evaluation of the Optimizers" with a focus on concurrency. For small files, pipelining is another dominant parameter especially when RTT is high which can be inferred by comparing Figure 2(a) with Figure 2(d). Parallelism is the second most dominant parameter for 100MB and 10GB files as it fulfills buffer requirement when TCP buffer size is less than BDP (e.g happens in XSEDE network as given in Table 1) and provides disk striping if the end-system makes use of disk striping on its storage system.

For stated reasons, instead of setting the same parameter combination for all files in a mixed dataset, we partition the dataset into chunks based on the file size and the Bandwidth Delay Product (BDP), and use different parameter combination for each chunk.

Algorithm 2 Single-Chunk Concurrency (SCC)

```
Require: source url, destination url, bandwidth, roundtrip time and maximum allowed buffer size
 1: function TRANSFER(source,destination,BW,RTT,BufferSize)
 2:     BDP = BW*RTT
 3:     allFiles = fetchFilesFromServer()
 4:     partitions = partitionFiles(allFiles,BDP)
 5:     while $partitions.count() > 0$ do
 6:         $Partition\ p = partitions.pop()$
 7:         $parameters = getOptimalParameters(p, BDP)$
 8:         transfer(p,parameters)
 9:     end while
10: end function
```

Algorithm 3 Multi-Chunk Concurrency (MCC)

```
Require: source url, destination url, bandwidth, roundtrip time and maximum allowed buffer size
 1: function TRANSFER(source,destination,BW,RTT,BufferSize)
 2:     BDP = BW*RTT
 3:     allFiles = fetchFilesFromServer()
 4:     partitions = partitionFiles(allFiles,BDP)
 5:     if partitions has HUGE$LARGE partition then
 6:         allocateChannel(HUGE,LARGE,1);    ▷ Allocate a channel for huge&large chunk
 7:         if partitions.contains SMALL&MIDDLE partition then
 8:             allocateChannels(SMALL,MIDDLE,concurrency-1);    ▷ If there exist small&middle size chunk then allocate rest of channels
 9:         else
10:             allocateChannel(LARGE,HUGE,1);
11:         end if
12:     else
13:         allocateChannel(SMALL,MIDDLE,concurrency)    ▷ if there is no large chunk, then share given channels within small chunks
14:     end if
15:     while $partitions.count() > 0$ do
16:         $Partition\ p = partitions.pop()$
17:         if $p.allocatedChannel > 0$ then
18:             transfer(partitions.get(i))
19:         end if
20:     end while
21: end function
```

As shown in Algorithm 1, we initially partition files into different chunks, then we check if each chunk has sufficient number of files using the *mergePartitions* subroutine. We merge a chunk with another if it is deemed to be too small to be a separate chunk. After partitioning files, we calculate the optimal parameter combination for each chunk. Pipelining and concurrency are the most effective parameters for small files to overcome poor utilization, so it is especially important to choose the best pipelining and concurrency values for small file transfers. We set the pipelining values by considering BDP and average file size of each chunk (lines 25, 29, 33 and 37); set the parallelism values by considering BDP, average file size, and the TCP buffer size (lines 26, 30, 34, and 38); and set the concurrency values by considering BDP, average file size, number of files in each chunk, and the maximum concurrency level (lines 27, 31, 35, and 39) in Algorithm 1.

As the average file size of a chunk increases, we decrease the pipelining value since it does not improve performance any more, and rather it can cause performance degradation by decreasing effects of concurrency. The method of selecting parallelism prevents using unnecessarily large parallelism level for small files and insufficiently small parallelism level for large files. Concurrency is set to larger values for small files, whereas it is limited to smaller values for large files,

Specs	XSEDE (Gordon & Lonestar)	LONI (Quenbee & Painter)
Bandwidth	10 Gbps	10 Gbps
RTT	60 ms	10 ms
TCP Buffer Size	32 MB	16 MB
BDP	75 MB	9 MB

Table 1: Network specifications of test environments

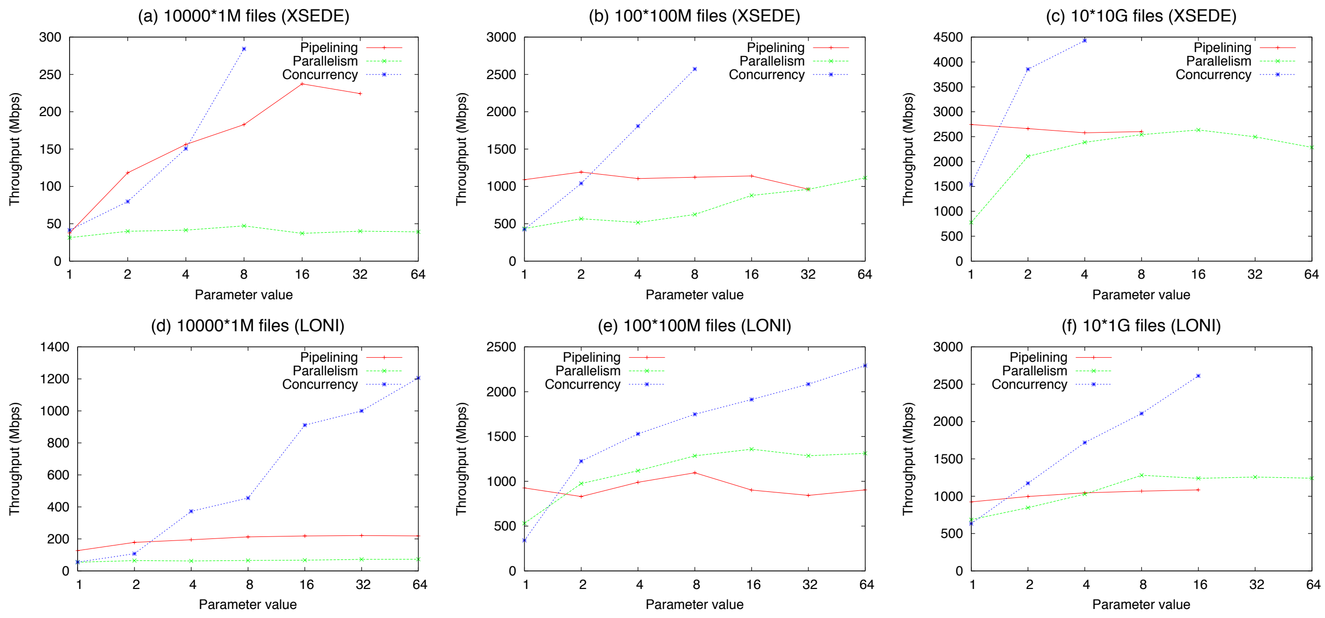

Figure 2: The effect of pipelining, parallelism, and concurrency on throughput

as higher concurrency values might cause unfair share of end-system and network resources. After obtaining optimal parameters for each chunk, SCC transfers each chunk separately as shown on lines 5-8 of Algorithm 2.

3.2 Multi-Chunk Concurrency (MCC)

In the Multi-Chunk Concurrency (MCC) method, the focus is mainly on minimizing the effect of small file chunks on the overall throughput. Based on Figure 2 and the SCC test results, we deduced that even after choosing the best parameter combination for each chunk, throughput obtained during the transfer of the small file chunks (named as SMALL and MIDDLE in Algorithm 1) is significantly worse compared to large chunks (named as LARGE and HUGE in Algorithm 1). Depending on the weight of small files' size over the total dataset size and concurrency level, overall throughput can be much less than the throughput of the large file chunk transfers. Thus, we developed the Multi-Chunk Concurrency (MCC) method which aims to minimize the effect of low throughput caused by the small files.

The idea behind MCC is to transfer multiple chunks concurrently on different channels, where one of the chunks is small and the other one is large so that small chunks do not affect the overall throughput significantly. Although we partitioned files into four chunks in Algorithm 1 to be more specific on parameter selection, SMALL and MIDDLE chunks follow a similar pattern in terms of poor transfer performance compared to LARGE and HUGE chunks. The MCC algorithm also considers fairness in resource sharing, so it first calculates how many concurrent processes are needed for each chunk using Algorithm 1. Then, if both small and large chunks exist, it opens only one channel for large chunks and uses rest of the available channels for small chunks as implemented on lines 5-9. Otherwise channels are shared between small and large chunks as shown on lines 10,13. The purpose of this is to achieve high performance without violating fairness policies too much. Furthermore, we do not close a channel after a transfer has completed on it. Instead, we recycle the channels to help other chunks which are either waiting to be run or are running slowly.

4. PERFORMANCE EVALUATION

In this section we present the performance evaluation results for the two StorkCloud optimizers we have introduced in the previous section, as well as the performance and scalability of the Directory Listing Service (DLS).

4.1 Performance of Optimizers

Figure 3 shows the effect of combining pipelining, parallelism and concurrency parameters on the transfer of different file sizes. The parameters in these experiments are chosen in a way that the best performing parameter values in the prior experiment are used as a fixed value in the successive experiment where other parameters are changed. In other words, we first tested effect of pipelining for each files size given in Figure 3(a) and 3(d), then we fixed pipelining values in Figure 3(b) and 3(e) to the best values obtained from Figure 3(a) and 3(d). We fixed both pipelining and parallelism in a similar way in Figure 3(c) and 3(f) when observing the concurrency effect. Although Figure 3 is not

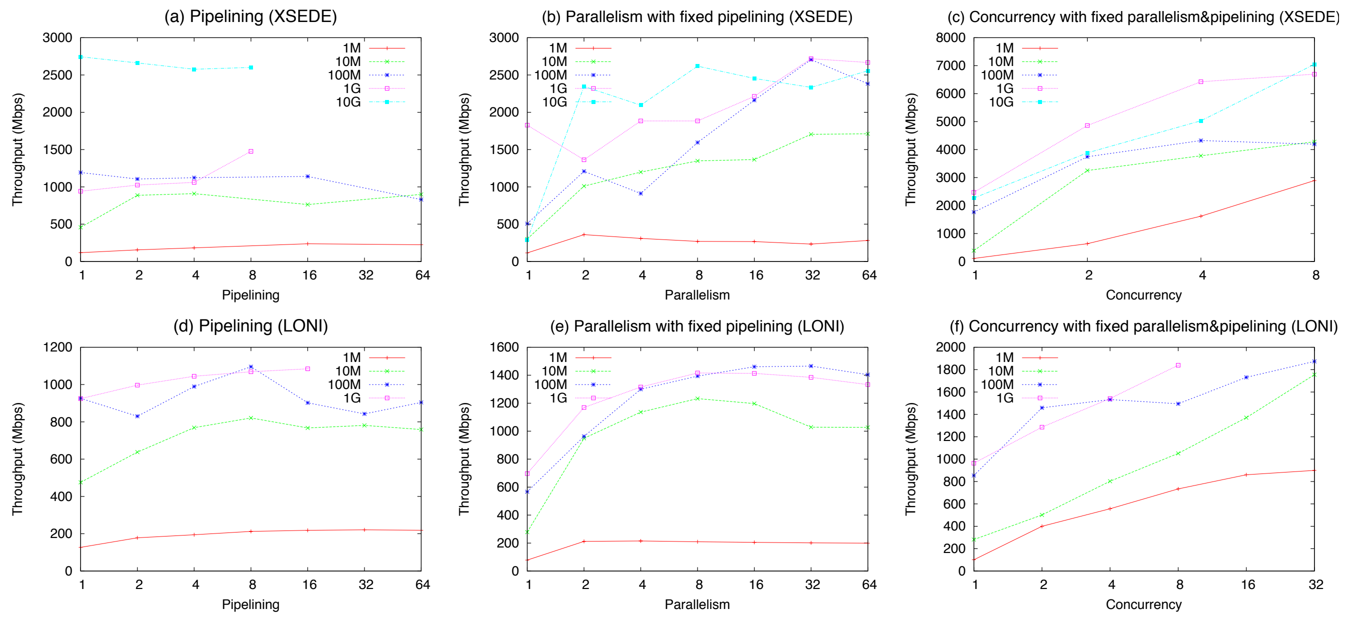

Figure 3: Effect of combining parameters on throughput

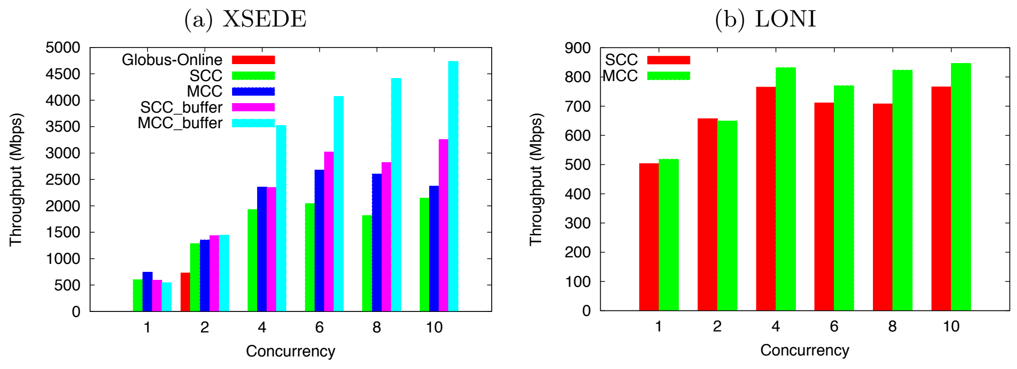

Figure 4: StorkCloud transfer performance on XSEDE and LONI

the optimal way of choosing parameter values, it gives a good approximation to the best possible value without running all possible parameter combinations.

We tested our algorithms on XSEDE and LONI testbeds, with specifications shown in Table 1. We used a mixed file size dataset with approximately 11,000 files with a total file size of about 140 GB. We ran two versions of our algorithms which are differentiated by either setting or not setting the TCP buffer size in transfers. Also, we compared their performance using up to 10 concurrency level as opening too many channels is neither fair nor allowed by testbed policies. We compared the performance of our algorithms with Globus Online when the concurrency is fixed to two, as Globus Online uses fixed two channels for all chunks it creates [5].

In Figure 4(a), we can see that both optimizers outperform Globus Online even when concurrency is fixed to two. Moreover, setting the buffer size to the maximum allowed value affects the throughput considerably when concurrency is set to more than two. Even though setting the buffer size improves the performance of large chunks regardless of concurrency level, when concurrency is less than four, small chunks dominate to overall throughput. As concurrency increases, small files are transferred faster, so overall throughput improves significantly.

The SCC optimization method performs better than the MCC in almost all cases. When concurrency is set to two or less, the difference is not too much due to the effect of many small files. However, the difference is more apparent when the concurrency level is more than two as it is able to suppress effect of small files poor performance effectively.

Results obtained on the LONI testbed are slightly different than XSEDE testbed. While increasing the concurrency level improves throughput, we were unable to see a sharp bounce in LONI tests as we observed in Figure 3 that concurrency does not improve throughput as much as it does in XSEDE network. This is basically because the disk read/write speed in LONI systems is lower than that of systems in the XSEDE testbed. Also, we observed that TCP slow start phase takes too long on the LONI testbed even though the round trip time is around 10 ms. Thus, transfer throughput cannot reach high values unless using very large files, such as 50GB or more.

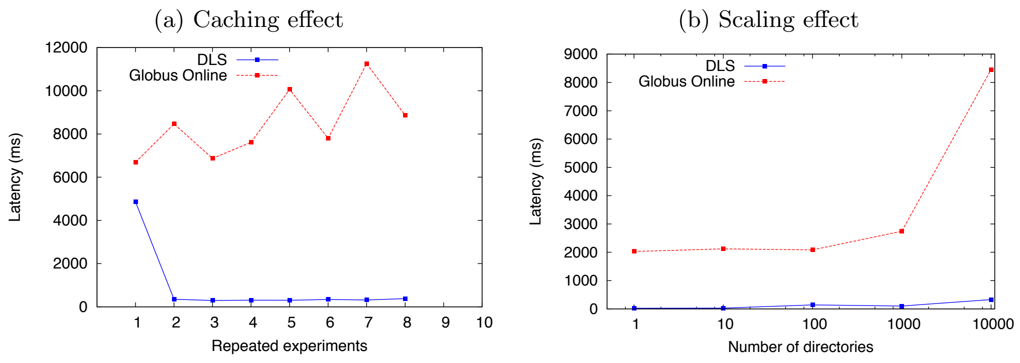

Figure 5: DLS versus Globus Online metadata access times

4.2 Performance of DLS

StorkCloud's Directory Listing Service (DLS) enables prefetching and caching of remote directory metadata in the Cloud to minimize response time to the users when traversing remote directory trees. In order to test the performance and scalability of DLS, we deployed the DLS server on Amazon EC2 and compared its directory traversal access times (latency) with Globus Online. We ran our thin client on DidcLab@UB and the remote directories to be traversed were located on Trestles@XSEDE. In Figure 5(a), we see the results of accessing and listing 1000 directories through StorkCloud DLS versus Globus Online. The first time these directories are traversed, StorkCloud responds slightly faster to the user's query compared to Globus Online, since these directories are not in the cache of DLS yet. But, if the same user or any other user request access to any portion of that directory tree again, the requested metadata will be in the cache of DLS and access through StorkCloud will be an order of magnitude faster compared to Globus Online.

Figure 5(b) shows the scaling effect when a user requests to access and traverse a directory tree with different number of subdirectories in it (i.e. 1, 10, 100, 1000, and 10000 respectively). The latency given in this figure is average value of multiple access requests for each data point. With the increased number of directories to traverse, we see the prefetching and caching effect of StorkCloud to be more effective and scalable compared to Globus Online.

5. RELATED WORK

The effort most comparable to our work is Globus Online [5], which provides data management as SaaS and offers fire-and-forget file transfers through thin clients (such as Web browsers) over the Internet. However, Globus Online only performs transfers between GridFTP servers, does not provide any throughput prediction capabilities, and its transfer optimizations are mostly done statically [5]. The data throughput estimation and optimization services provided by StorkCloud can feed information to data transfer services such as Globus Online, allowing for dynamic transfer optimization and thus more efficient transfers.

Liu et al. [28] developed a tool which optimizes multi-file transfers by opening multiple GridFTP threads. The tool increases the number of concurrent flows up to the point where transfer performance degrades. That work only focuses on using threading to increase performance when transferring files; other transfer parameters (e.g., buffer size and parallel streams) are not optimized. One issue with the technique is, in cases where the average file size is very small, hundreds of threads may be needed, which might result in system instability.

Chen et al. [13] offer a number of solutions to optimize bulk data transfer between data centers motivated by the fact that the dominant component of inter-datacenter traffic is background bulk data traffic. Other approaches aim to improve throughput by opening multiple flows over multiple paths between source and destination [30, 20, 35], however there are cases where individual data flows fail to achieve optimal throughput because of end-system bottlenecks. Several others propose solutions that improve utilization of a single path by means of parallel streams [6, 18, 29, 36, 9], pipelining [3, 16, 10], and concurrent transfers [28, 26, 24]. Although using parallelism, pipelining, and concurrency may improve throughput in certain cases, an optimization algorithm [34, 28, 5] should also consider system configuration, since end-systems may present factors (e.g., low disk I/O speeds or over-tasked CPUs) which can introduce bottlenecks.

Some components of our system (e.g., DLS) take advantage of prefetching and caching in order to reduce access time and improve responsiveness. These techniques have been widely studied in the past in different contexts [11, 12, 32, 8]. Our work aims to offer a single logical-view proxy for caching and serving directory metadata information as a Cloud service. In a proxy caching system such as ours, responses from remote servers to metadata requests are cached and periodically updated. The cached responses are served to users who request the same data, reducing latency and network load.

6. CONCLUSIONS AND FUTURE WORK

We have presented the initial design and prototype implementation of StorkCloud, which mitigates the large-scale end-to-end data movement bottleneck by efficiently utilizing underlying networks and effectively scheduling and optimizing data transfers. It provides a Cloud-hosted data scheduling and optimization service with enhanced functionality such as data aggregation and connection caching; early error detection and recovery; scheduled storage management; and end-to-end performance optimization services which will benefit a diverse set of data-intensive Cloud computing applications.

Some upcoming features include automatic transfer module chaining and temporary data parking in the Cloud. Module chaining will allow transfers between arbitrary protocols by using one transfer module to retrieve data and simultaneously using another to send it. Data parking will allow transfers that might fail only at the receiving end to be resumed without needing further contact with the sending end by having the Cloud store file contents until the receiving end becomes active again. We also plan on implementing a number of caching and prefetching heuristics in DLS and our smartphone and Web clients that aim to improve interface responsiveness and overall user experience.

7. ACKNOWLEDGMENTS

This project is in part sponsored by NSF under award numbers CNS-1131889 (CAREER), OCI-0926701, (STCI-Stork), and CCF-1115805 (CiC-Stork).

8. REFERENCES

[1] Arra/ani testbed. https://sites.google.com/a/lbl.gov/ani-100g-network.

[2] Classified advertisements (classads). http://research.cs.wisc.edu/htcondor/classad/classad.html.

[3] SMTP service extension for command pipelining. http://tools.ietf.org/html/rfc2920.

[4] A Toroidal LHC ApparatuS Project (ATLAS). http://atlas.web.cern.ch/.

[5] Allen, B., Bresnahan, J., Childers, L., Foster, I., Kandaswamy, G., Kettimuthu, R., Kordas, J., Link, M., Martin, S., Pickett, K., and Tuecke, S. Software as a service for data scientists. *Communications of the ACM 55:2* (2012), 81–88.

[6] Altman, E., and Barman, D. Parallel tcp sockets: Simple model, throughput and validation. In *INFOCOM* (2006).

[7] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. Basic Local Alignment Search Tool. *Journal of Molecular Biology 3*, 215 (October 1990), 403–410.

[8] Bala, K., Kaashoek, M. F., and Weihl, W. E. Software prefetching and caching for translation lookaside buffers. In *Proc. of OSDI 2004*.

[9] Balman, M., and Kosar, T. Dynamic adaptation of parallelism level in data transfer scheduling. In *Proc. of CISIS 2009*.

[10] Bresnahan, J., Link, M., Kettimuthu, R., Fraser, D., and Foster, I. Gridftp pipelining. In *TeraGrid* (2007).

[11] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. A study of integrated prefetching and caching strategies. *SIGMETRICS Perform. Eval. Rev. 23*, 1 (May 1995), 188–197.

[12] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst. 14*, 4 (Nov. 1996), 311–343.

[13] Chen, Y., Jain, S., Adhikari, V. K., Zhang, Z.-L., and Xu, K. A first look at inter-data center traffic characteristics via yahoo! datasets. In *INFOCOM* (2011), IEEE, pp. 1620–1628.

[14] Cho, B., and Gupta, I. Budget-constrained bulk data transfer via internet and shipping networks. In *Proc. of ICAC* (2011).

[15] CMS. The US Compact Muon Solenoid Project. http://uscms.fnal.gov/.

[16] Farkas, K., Huang, P., Krishnamurthy, B., Zhang, Y., and Padhye, J. Impact of tcp variants on http performance. *Proc. of High Speed Networking* (2002).

[17] Garfienkel, S. An evaluation of Amazonï£¡s Grid computing services: EC2, S3 and SQS. Tech. Rep. TR-08-07, Aug 2007.

[18] Hacker, T. J., Noble, B. D., and Athey, B. D. Adaptive data block scheduling for parallel tcp streams. In *Proc. of HPDC* (2005).

[19] Hacker, T. J., Noble, B. D., and Atley, B. D. The end-to-end performance effects of parallel tcp sockets on a lossy wide area network. In *Proc. of IPDPS* (2002).

[20] Khanna, G., Catalyurek, U., Kurc, T., Kettimuthu, R., Sadayappan, P., Foster, I., and Saltz, J. Using overlays for efficient data transfer over shared wide-area networks. In *Proc. of SC* (2008).

[21] Kiehl, J., Hack, J. J., Bonan, G. B., Boville, B. A., Williamson, D. L., and Rasch, P. J. The national center for atmospheric research community climate model: Ccm3. *Journal of Climate 11:6* (1998), 1131–1149.

[22] Kim, J., Yildirim, E., and Kosar, T. A highly-accurate and low-overhead prediction model for transfer throughput optimization. In *Proc. of DISCS Workshop* (2012).

[23] Klein, R. J. T., Nicholls, R. J., and Thomalla, F. Resilience to natural hazards: How useful is this concept? *Global Environmental Change Part B: Environmental Hazards 5*, 1-2 (2003), 35 – 45.

[24] Kosar, T., and Balman, M. A new paradigm: Data-aware scheduling in grid computing. *Future Generation Computing Systems 25*, 4 (2009), 406–413.

[25] Kosar, T., Balman, M., Yildirim, E., Kulasekaran, S., and Ross, B. Stork data scheduler: Mitigating the data bottleneck in e-science. *The Phil. Transactions of the Royal Society A 369(3254-3267)* (2011).

[26] Kosar, T., and Livny, M. Stork: Making data placement a first class citizen in the grid. In *Proc. of ICDCS* (2004).

[27] Laoutaris, N., Sirivianos, M., Yang, X., and Rodriguez, P. Inter-datacenter bulk transfers with netstitcher. In *ACM SIGCOMM* (2011).

[28] Liu, W., Tieman, B., Kettimuthu, R., and Foster, I. A data transfer framework for large-scale science experiments. In *Proc. of DIDC Workshop* (2010).

[29] Lu, D., Qiao, Y., Dinda, P. A., and Bustamante, F. E. Modeling and taming parallel tcp on the wide area network. In *Proceedings of IPDPS '05* (April 2005), IEEE, p. 68.2.

[30] Raiciu, C., Pluntke, C., Barre, S., Greenhalgh, A., Wischik, D., and Handley, M. Data center networking with multipath tcp. In *Proc. of ACM HotNets-IX* (2010).

[31] Research, F. The Future of Data Center Wide-Area Networking. info.infineta.com/l/5622/2011-01-27/Y26.

[32] Voelker, G. M., Anderson, E. J., Kimbrel, T., Feeley, M. J., Chase, J. S., Karlin, A. R., and Levy, H. M. Implementing cooperative prefetching and caching in a globally-managed memory system. *SIGMETRICS Perform. Eval. Rev. 26*, 1 (June 1998), 33–43.

[33] Yildirim, E., Kim, J., and Kosar, T. Optimizing the sample size for a cloud-hosted data scheduling service. In *Proc. of CCSA Workshop* (2012).

[34] Yildirim, E., and Kosar, T. Network-aware end-to-end data throughput optimization. In *Proc. of NDM 2011*.

[35] Yildirim, E., Suslu, I., and Kosar, T. Which network measurement tool is right for you? A multidimensional comparison study. In *Proceedings of GRID 2008*.

[36] Yildirim, E., Yin, D., and Kosar, T. Balancing tcp buffer vs parallel streams in application level throughput optimization. In *Proc. of DADC Workshop* (2009).

[37] Yildirim, E., Yin, D., and Kosar, T. Prediction of optimal parallelism level in wide area data transfers. *IEEE TPDS 22(12)* (2011).

[38] Yin, D., Yildirim, E., and Kosar, T. A data throughput prediction and optimization service for widely distributed many-task computing. *IEEE TPDS 22(6)* (2011).