An Elegant Sufficiency: Load-Aware Differentiated Scheduling of Data Transfers

Rajkumar Kettimuthu
Argonne National Laboratory
kettimut@anl.gov

Gayane Vardoyan
University of Massachusetts
gvardoyan@cs.umass.edu

Gagan Agrawal
The Ohio State University
agrawal.28@osu.edu

P. Sadayappan
The Ohio State University
sadayappan.1@osu.edu

Ian Foster
Argonne National Laboratory
foster@anl.gov

ABSTRACT

We investigate the file transfer scheduling problem, where transfers among different endpoints must be scheduled to maximize pertinent metrics. We propose two new algorithms that exploit the fact that the aggregate bandwidth obtained over a network or at a storage system tends to increase with the number of concurrent transfers—but only up to a certain limit. The first algorithm, SEAL, uses runtime information and data-driven models to approximate system load and adapt transfer schedules and concurrency so as to maximize performance while avoiding saturation. We implement this algorithm using GridFTP as the transfer protocol and evaluate it using real transfer logs in a production WAN environment. Results show that SEAL can improve average slowdowns and turnaround times by up to 25% and worst-case slowdown and turnaround times by up to 50%, compared with the best-performing baseline scheme. Our second algorithm, STEAL, further leverages user-supplied categorization of transfers as either "interactive" (requiring immediate processing) or "batch" (less time-critical). Results show that STEAL reduces the average slowdown of interactive transfers by 63% compared to the best-performing baseline and by 21% compared to SEAL. For batch transfers, compared to the best-performing baseline, STEAL improves by 18% the utilization of the bandwidth unused by interactive transfers. By *elegantly* ensuring a *sufficient*, but not excessive, allocation of concurrency to the right transfers, we significantly improve overall performance despite constraints.

1. INTRODUCTION

Data volumes in science are growing exponentially, a trend that is expected to continue and even accelerate. For example, state-of-the-art detectors at light sources [10] generate tens of terabytes of data per day, and future camera-storage bus technologies may increase data rates by two orders of magnitude. Genomic data sets are growing faster than Moore's law [36]. The internationally distributed Earth System Grid Federation (ESGF) [53] provides access to more than 1 PB of climate simulation and observation data—a volume that will grow by orders of magnitude over the next decade.

While increased data volumes pose challenges for many areas of computer science, one issue that has received less attention is the allocation of the multiple resources typically involved in end-to-end data movement. Large datasets must often be transported over wide-area networks (WANs) for analysis, visualization, or archiving. A single wide-area transfer may involve many resources at the source, destination, and intervening points, including disks, storage area networks (SANs), file systems, data transfer nodes (DTNs) [22], SAN and WAN interfaces on DTNs, and campus networks, in addition to WAN links. These resources are all typically shared, with disks, SANs, and file systems used by both file transfers and local compute jobs. Campus network and WAN resources are shared by file transfers that traverse associated DTNs and by other external traffic that uses these network links. And although network backbones are often overprovisioned, the same may not be true of all resources in the end-to-end path. Depending on system configuration, any resource involved in a WAN data movement can be a bottleneck. Also, given predictions of science traffic's exponential growth, several recent studies [6, 9, 23] project that network overprovisioning may not continue. The frequently bursty nature [47, 50] of science traffic introduces further challenges.

Despite much work on accelerating individual file transfers (e.g., [31, 48, 49, 56]), the scheduling of multiple transfers to improve both aggregate performance and the performance of individual flows has received little attention [27]. In contrast, work on the conceptually similar problem of executing parallel computational jobs efficiently on supercomputers has motivated decades of work on *job scheduling* [1, 15, 16, 24, 40, 46], and the development of sophisticated job scheduling algorithms that can optimize schedules while also addressing other goals such as minimizing average slowdown [25] and turnaround time, for a wide range of job types. The need for rapid response in many applications means that *relative slowdown* is often more important to users than absolute delay. For example, a user will likely be less happy about a 10-second delay to a 10-second task than to a 10-hour task. One can reduce average relative slowdown by selectively delaying longer tasks.

The corresponding *transfer scheduling* problem introduces interesting new challenges due to important differences between computational jobs and file transfers, including the following. First, in contrast to nodes on a supercomputer, the shared resources involved in an end-to-end transfer are often not managed explicitly as schedulable resources. Thus, alternative enforcement methods are required, based for example on admission control of requests rather than explicit resource allocation. Second, resources may be subject to arbitrary *external load* that is neither under our control nor directly visible to us; thus we must use different methods for determining resource availability, such as monitoring of historical and recent performance. Third, the aggregate bandwidth achieved over a network or at a parallel storage system is typically greater when multiple transfers occur at the same time [35, 37]. Thus, it can be advantageous to schedule multiple transfers at once—or, if a sys-

SC '15, November 15 - 20, 2015, Austin, TX, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-3723-6/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807660

tem is not saturated, to divide large files into multiple chunks that can be communicated concurrently. Fourth, the benefits from such increased concurrency do not grow beyond a threshold; indeed, all transfers ultimately experience slowdown when concurrency is too high [38, 57]. Thus, when one or more resources involved in data transfers is/are heavily loaded, stalling some transfers until other transfers finish can reduce average turnaround time.

We leverage these properties of transfers in two new algorithms for the efficient scheduling of a set of transfers. The first is a load-aware scheduling algorithm (*SchEduler Aware of Load*: `SEAL`) that adaptively schedules file transfers using a strategy that monitors external load, and controls scheduled load, in order to minimize average slowdown across all transfers. This algorithm preempts and/or delays transfers when in so doing it can reduce average slowdown (for example, under heavy load). It increases the concurrency used for a particular file transfer when in so doing it can increase aggregate performance (for example, under low load).

The second algorithm (*Scheduler TypE Aware and Load aware*: `STEAL`) supports differential treatment of two different transfer types. What we call *interactive* transfers (e.g., remote analysis of simulation and observational data) must be completed as soon as possible; these transfers require best-effort service. In contrast, *batch* transfers (e.g., certain data replication, backup, archiving tasks) have more flexibility [7, 8, 47]; they may need only to be completed within a certain window (e.g., 24 hours)—a window that may be several times longer than the transfer time under average load. `STEAL` extends `SEAL` to enable *batch* transfers to use bandwidth unused by *interactive* transfers, with minimal impact (in terms of *slowdown*) on *interactive* transfer performance.

We experimentally evaluate `SEAL` and `STEAL` in a production WAN environment comprising high-end computers at geographically disparate locations. In these experiments, we replay transfers from real transfer logs that capture the large variations (in the size and achievable bandwidth of individual transfers, and number of transfers over time) that are typical of high-end network environments. We also add synthetic *batch* transfers when evaluating `STEAL`. We find that `SEAL` can improve average slowdown and turnaround time by up to 25%, and worst-case slowdown and turnaround time by up to 50%, relative to an efficient baseline scheme. We also determine that when *batch* tasks are introduced, `STEAL` reduces the average slowdown of interactive transfers by 63% compared to the best-performing baseline and by 21% compared to `SEAL`. For batch transfers, it improves their usage of the bandwidth unused by interactive transfers by 18% compared to the best-performing baseline, while roughly matching `SEAL` (1.75% degradation) on that metric.

2. MOTIVATION

We elaborate on two WAN transfer features that inform this work.

2.1 Endpoint Load Varies Greatly

We studied Globus GridFTP usage logs [30] for different 24-hour periods for the 10 sites that transferred the most bytes during those periods (we chose GridFTP because it is widely used for bulk data transfer [12, 21]). This analysis revealed that traffic is typically nonuniform, with bursts saturating system resources. Figure 1 shows the number of concurrent transfers over one 24-hour period at a busy site. During this period, the number of concurrent transfers ranges from 0 to more than 100, with a mean of 45.7 and standard deviation of 30.7. We observed similar trends in logs for other 24-hour periods and sites, with the standard deviation of the number of concurrent transfers ranging between 53% and 141% of the mean. We conclude that any effective scheduling algorithm must be able to deal with widely varying load levels.

2.2 Different Transfers Have Different Needs

While certain transfer requests must be processed rapidly, others

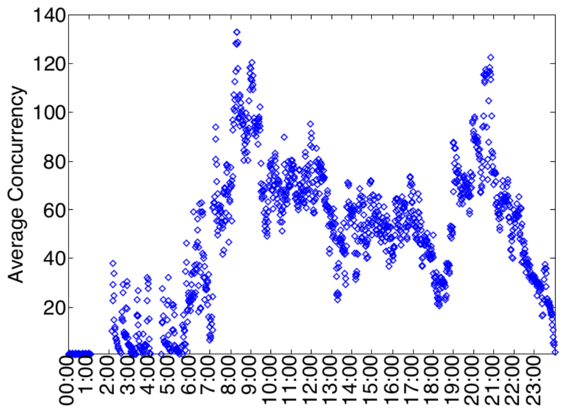


Figure 1: Distribution of file transfers for a loaded server. X-axis: time in hours. Y-axis: average concurrency per 60-second interval.

can tolerate larger delays. Recent reports on science network requirements [8, 11, 34] give replication as a common, often relatively time-insensitive, reason for moving large quantities of data, whether for performance [43, 53], fault tolerance [7], and/or preservation [33]. Another motivator for delay-tolerant large-scale data movement can be changes in storage system availability, such as a storage system reaching capacity or shutting down, or a storage allocation expiring [8].

As science data typically does not change rapidly, large transfers for updating a replica can often tolerate delays. For example, the cited reports describe replication use cases where TB datasets must be delivered overnight. Because subsequent processing involves manual steps, there is no advantage in completing the transfer earlier. A terabyte of data can be transferred in under 45 minutes at 3 Gbps—a disk-to-disk WAN transfer rate that is commonly achieved between endpoint pairs in today's HPC environment. Thus, transfer times can vary by at least an order of magnitude without compromising science goals. Other use cases described in the reports have similar ranges, with one requiring only that 100 TB be transferred "within a month."

Other well-known use cases involve *interactive* jobs. For example, when a user requests a (set of) file(s) from a resource such as ESGF with the goal of visualizing this data, they want the download to be completed as soon as possible. The reports cited above describe time-sensitive use cases in which data generated at experimental facilities and computer simulations must be moved to a remote data analysis facility as soon as possible—for example, to permit comparison, visualization, and/or validation.

We classify those transfers for which it is acceptable for transfer times to be significantly (an order of magnitude) longer than average as *batch* transfers, and all other transfers as *interactive* transfers. We will leverage the flexibility of *batch* transfers to improve the performance of *interactive* transfers, while also maximizing spare bandwidth utilization for *batch* transfers.

3. DATA TRANSFER SCHEDULING

We next discuss the challenges in scheduling wide-area transfers and the approach we use to address these challenges.

3.1 Challenges in Scheduling

In parallel job scheduling, the total time needed to execute an uninterrupted job is largely unaffected by other environmental factors (I/O contention is an exception). In contrast, the time required for wide-area file transfers can vary significantly due to interference among different, often highly dynamic network flows.

The use of multiple TCP streams and concurrent file transfers is often required in order to achieve file transfer rates close to network speeds [35, 56]. However, indiscriminately increasing the number of concurrent transfers for all files is rarely effective, because capabilities at different sites can differ widely with respect to network interface card (NIC) capacity, storage speed, CPU resources, and WAN connectivity. Indeed, a powerful source that attempts to push data too rapidly to a less powerful destination can greatly degrade aggregate throughput—for example, performance declining by as much as 40x when a buffer in an intervening router overflows [13].

We thus need adaptive methods that will increase concurrency only for destinations that have not reached their physical limit. When load is low, such methods can increase concurrency so as to maximize utilization. When load increases, they should not schedule new requests if in so doing total concurrency is increased above a useful limit: instead, they should delay requests and/or reduce the concurrency of ongoing transfers. Such methods can increase aggregate end-to-end throughput while also ensuring that the response times of individual transfer tasks remain reasonable. The bursty nature of wide-area transfers provides significant opportunities for such methods to improve both aggregate throughput and average slowdown.

The current state of the art in file transfer scheduling is best-effort. Each arriving transfer is scheduled unless a system-specified concurrency limit is reached. For example, Globus transfers a maximum of two files concurrently with either two or four TCP streams (depending on average file size) for each transfer [4]. This approach has two disadvantages. First, under heavy load the completion times of all transfer tasks can suffer, increasing average slowdown. In comparison, scheduling only as many tasks as needed to saturate the system can reduce completion time for those tasks without much increase in completion time for other transfers. Second, not increasing concurrency when the number of pending transfers is small can reduce overall utilization. We address these limitations via an algorithm that queues, preempts, and dynamically adjusts transfer concurrency.

3.2 Metrics

We define distinct metrics for *interactive* and *batch* transfers.

3.2.1 Interactive File Transfers

In (compute) scheduling, average *response time* or *turnaround time* (*completion time – arrival time*) has long been used as a measure of scheduler quality [25]. More recently, *job slowdown* or *stretch* has emerged as a more suitable measure. Job slowdown is the factor by which a job is slowed relative to the time it would take on an unloaded system. This metric has been widely used to study the performance of parallel job schedulers [25] and databases [45]. We choose (a variant of) it as the optimization metric in our work. More specifically, we start with the *bounded slowdown* or *BS* metric [25]. This metric, first introduced in the context of parallel job scheduling, seeks to limit the influence of extremely short jobs on slowdown by measuring the slowdown of such jobs relative to an *interactive threshold* or *bound*, rather than the actual runtime:

$$BS = \frac{\text{Waittime} + \max(\text{Runtime}, \text{bound})}{\max(\text{Runtime}, \text{bound})}. \qquad (1)$$

To measure the quality of our scheduler for *interactive* transfers, we further adapt the *bounded slowdown* metric. Specifically, we modify the definition of BS above to account for the fact that in the file transfer context, the *runtime* is not necessarily constant: the time it takes to move a file from source to destination can vary according to other loads on the system, because all resources involved (source storage devices, source hosts, networks, destination hosts, destination storage devices) are shared. In contrast, in traditional parallel job scheduling, nodes are used in a dedicated fashion and thus the *runtime* of a job on a given number of nodes can typically be treated as fixed. Thus, we define bounded slowdown for a file transfer task, BS_{FT}, as follows:

$$BS_{FT} = \frac{\text{Waittime} + \max(\text{Runtime}, \text{bound})}{\max(TT_{ideal}, \text{bound})}, \qquad (2)$$

where TT_{ideal} is the estimated transfer time (TT) under zero load and ideal concurrency. We use the method shown in Listing 2 to compute TT_{ideal}. This method is based on a model described in our previous work [38] (summarized in §4.2). In the work reported here, we set the *bound* to one second. In the rest of this paper, we refer to BS_{FT} as *slowdown*.

3.2.2 Batch File Transfers

Since *batch* transfers can tolerate longer delays, it is acceptable to delay them relative to *interactive* transfers. Thus, *bounded slowdown* by itself is not a suitable metric for *batch* transfers. Instead, we define a bi-objective scheduling problem. First, as we want *batch* transfers to use as much unused bandwidth as possible, we focus on the fraction of the spare bandwidth used by *batch* transfers. More specifically, if B_T is the total bandwidth available, B_I is the bandwidth consumed by *interactive* jobs, and B_B is the bandwidth used for *batch* jobs, we aim to maximize $\frac{B_B}{B_T - B_I}$.

A scheduler can maximize $\frac{B_B}{B_T - B_I}$ just by prioritizing *batch* jobs. Thus we must introduce a second objective. Suppose that there are are no *batch* jobs in the system, and the average slowdown for *interactive* jobs is SD_I. Next, the scheduler adds a set of *batch* jobs, but still prioritizes *interactive* jobs. Let the average slowdown for *interactive* jobs now be SD_{I+B}. Our second objective is to maximize $\frac{SD_I}{SD_{I+B}}$, i.e., to achieve a value as close to 1 as possible.

4. ADAPTIVE SCHEDULING

We formulate the load-aware scheduling problem (with interactive jobs only) and describe our approach and the `SEAL` algorithm.

4.1 Problem Formulation

We consider a stream of file transfer requests, each defined by a six-tuple: <*source host*, *source file path*, *destination host*, *destination file path*, *file size*, *arrival time*>. Requests arrive in an online fashion, i.e., future transfer requests are not known a priori. Hosts may have different capabilities (CPU, memory, disk speed, storage area network, network interfaces, WAN connection) and thus the maximum achievable end-to-end throughput may differ for each <*source host, destination host*> pair. Load at a source, destination, and intervening network may also vary over time, as may the achievable transfer rates between a source and destination. Each host (source or destination) has a limit on the number of concurrent transfers that it can support. The problem is to schedule transfers so as to minimize average transfer slowdown.

4.2 Throughput Estimation

To estimate the throughput for a given transfer, we leverage a model from our previous work [38]. This model combines extensive historical data with a correction term that accounts for current external load. It takes three pieces of input: first, a signature for a given transfer, encompassing its concurrency level, total known concurrency at source ("known load at source"), and total known concurrency at destination ("known load at destination"); second, historical data (transfer concurrency, known loads, and observed throughput) for the source-destination pair; and third, information (signatures and observed throughputs) from the most recent transfers for the source-destination pair. It produces an estimated throughput as an output. We augmented the signature and models in the work reported here to also include the transfer size.

This approach takes advantage of the fact that transfer load tends to be stable over short time periods (for example, 30 minutes) but can vary greatly over longer durations (for example, a day) [38]. We assume, furthermore, that we have access to historical data that includes many combinations of signature values. Inevitably, while this data will include recent transfers, we cannot expect to have recent transfers with all possible combinations of signature values.

In order to combine the copious historical data with the sparser

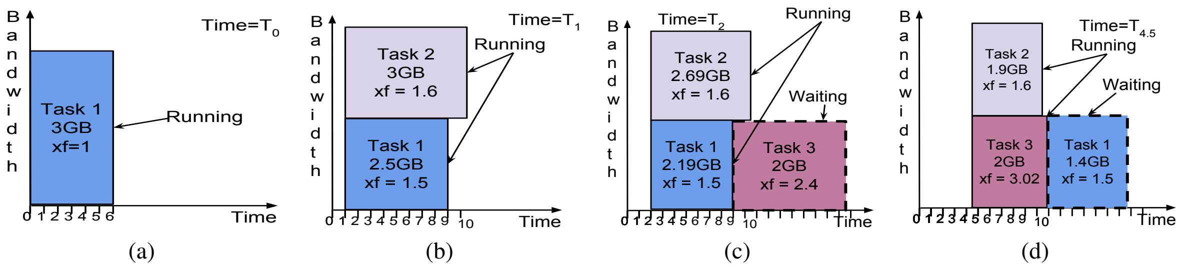


Figure 2: Example illustrating SEAL. The width of each task is its expected runtime (TT_{load}), and the height of stacked running tasks is their aggregate throughput. The text inside each box shows the amount of data remaining to be transferred and the task's *xfactor* at that time. *pf* is 2. (a) Task 1 arrives and is scheduled. (b) Task 2 arrives and is scheduled. (c) Task 3 arrives and is queued because the system is saturated. (d) Task 3's *xfactor* is large enough to preempt Task 1. Task 1 is moved to the wait queue, and Task 3 is scheduled.

recent data, we determine the average historical throughput for a particular source, destination, and signature; compute the difference between the throughput of recent transfers between the same source and destination (but likely different signatures) and the historical average for the corresponding transfer signature; and use this data to correct the bandwidth predicted by the historical data for throughput at the desired signature.

4.3 The SEAL Algorithm

SEAL is an online load-aware scheduling algorithm that schedules data transfers adaptively to reduce average slowdown. It (a) queues transfers so as to bound concurrency during high-load situations and (b) increases transfer concurrency during low-load situations.

Scheduling strategies that aim to reduce slowdown typically use an expansion factor (*xfactor*) parameter, the expected slowdown of a task at any given time, when prioritizing tasks. Similarly to how we modified the expression for *BS* to define BS_{FT} in §3.2.1, we define the expansion factor for a file transfer task, $xfactor_{FT}$, as

$$xfactor_{FT} = \frac{Waittime + TT_{load}}{TT_{ideal}}, \qquad (3)$$

where TT_{load} is the estimated transfer time (TT) under the current load conditions and TT_{ideal} is the estimated transfer time under ideal (zero load and ideal concurrency) conditions. Both TT_{load} and TT_{ideal} are computed based on the model described in §4.2 (see Listing 2). Note the difference between BS_{FT} and $xfactor_{FT}$: BS_{FT} is a metric used to evaluate how well a task performed; it requires the actual runtime of a task and can be computed only after the task is completed. In contrast, $xfactor_{FT}$ is an expected value, used as a priority value for tasks that have not yet completed and for which actual runtime is therefore unknown. In the rest of this paper, we use the term *xfactor* to refer to $xfactor_{FT}$.

Before providing a formal description of the SEAL algorithm, we explain the main ideas. Overall, it involves four decisions: (1) Should a new transfer be scheduled or queued? (2) When scheduling a transfer, what concurrency should be used? (3) When should an already scheduled transfer be preempted? (4) When should the concurrency of an ongoing file transfer be changed? In making these decisions, SEAL uses both the models described in §4.2 and the observed performance of current transfers, as follows.

First, SEAL always schedules waiting transfers if neither source nor destination are saturated, with our models to determine the concurrency for any transfer that is scheduled. We conclude that an endpoint E is saturated if either of the following is true. (a) Aggregate observed throughput for all transfers involving that endpoint is close ($\geq$95%) to the maximum possible throughput, as revealed by previous empirical measurements (or historical data); we maintain a moving five-second average observed throughput for each transfer for this purpose. (b) Increased concurrency results in a proportionately insignificant increase in estimated throughput on several active links involving that endpoint, meaning that if concurrency is increased by a factor F, throughput is increased only by a factor of $0.25 \times F$ or less. (More specifically, we test three source-destination pairs that are in current use, or less than three if fewer are active. The rationale here is that if increased concurrency results in little increase in throughput on a given link, then either one or both endpoints are likely saturated. If this behavior is observed on three links, E–X, E–Y, and E–Z, then it is likely that endpoint E is saturated.)

Second, if the source or destination is saturated, the algorithm can still interrupt one or more active transfer(s) to service waiting requests, if in so doing it can reduce overall average slowdown. As each such preemption incurs a connection establishment and authentication cost, and we must also retransmit data from the last checkpoint, we use a *preemption factor* (*pf*) to control the number of preemptions. *pf* is the minimum ratio of the *xfactor* of a waiting task to the *xfactor* of a running task for preemption to occur. Note that the inclusion of wait time when computing a task's *xfactor* (Equation 3) ensures that no task is delayed indefinitely.

Third, the algorithm dynamically increases the concurrency of ongoing transfers if two conditions are met. First, there must not be any queued transfers, and second, there is bandwidth available due to completion of transfers.

We use the example in Figure 2 to illustrate some key aspects of SEAL. We assume one source and one destination, each capable of a maximum throughput of 5Gbps. Task T1, a 3GB file, arrives at $t = 0$. Suppose that from the available 5Gbps bandwidth, a single task can use up to 4Gbps with a suitably high concurrency. Since no other tasks are running, T1's $TT_{load} = TT_{ideal} = 6$ secs, and its *xfactor* is 1. At $t = 1$, task T2, a 3GB file, arrives. T2 is scheduled immediately since neither source nor destination is saturated. The total available bandwidth is then split between the two tasks: let us say that it is 2.5Gbps for each, which results in a higher TT_{load} and *xfactor* for T1. At time $t = 2$, task T3, a 2GB file, arrives but is made to wait because the system is saturated. T3 has to wait until its *xfactor* becomes *pf* times the *xfactor* of one of the running tasks, at which point it can preempt the running task and get scheduled (we use a *pf* of 2 in this example). At $t = 4.5$, T3's *xfactor* becomes higher than *pf* times T1's *xfactor*. Hence, T1 is preempted, and T3 is activated. At $t = 4.5$, T2 has $\sim$1.9 GB left to transfer. At $t = 10.6$, T2 completes, and T1 is rescheduled. At $t = 10.9$, T3 completes. T1 has $\sim$1.3 GB left to transfer but can now return to its 4 Gbps transfer rate and thus completes at $t = 13.53$. Thus, the turnaround times for T1, T2, and T3 are 13.53, 9.6, and 8.9 seconds, respectively. The average turnaround time is 10.68 seconds.

For the same scenario, a default baseline algorithm that starts tasks as soon as they arrive would behave as SEAL until T3 arrives. T3 is scheduled upon arrival at time $t = 2$, and all tasks begin to transfer at 1.67Gbps. T1 completes at $t = 12.54$, T2 at $t = 14.95$, and T3 at $t = 11.6$. Thus, the turnaround times for the three tasks are 12.54, 13.95, and 9.6 sec, respectively: an average of 12.03, more than 12% worse than SEAL. Furthermore, this analysis assumes that the baseline and SEAL use the same concurrency; in practice, the baseline's static concurrency would often be sub-optimal, and thus its performance would be yet worse.

4.4 Formal Description of SEAL

We present pseudocode for SEAL in Listings 1 and 2. Table 1 describes the main data structures and terms. The function **throughput** estimates throughput as described in §4.2 and the function **saturated** determines whether an endpoint is saturated as described in §4.3. The scheduling cycle repeats every n seconds, and NT contains all new tasks that arrived in those n seconds. (In our implementation, $n = 0.5$.) At the start of each cycle, completed tasks are removed from the run queue R, new tasks are added to the wait queue W, and *xfactor* values are updated (Listing 1, lines 2 to 6).

Table 1: Summary of terms used in SEAL (and STEAL).

Item	Description
$task$	A transfer task
NT	Set of new tasks
R	Priority queue of running tasks (ascending *xfactor*)
W	Priority queue of waiting tasks (descending *xfactor*)
CL	Candidate list of tasks for preemption
WT	Wait time
cc	Concurrency
$size$	Transfer size
TT_{load}	Transfer time under current load (predicted by model)
TT_{ideal}	Ideal transfer time (predicted by model)
TT_{trans}	Time the task has not been idle so far
β	User-defined variable for increasing concurrency
$maxCC$	maximum concurrency allowed for a task
λ	User-defined fraction to limit *batch* bandwidth [STEAL]

Listing 1 One scheduling cycle for SEAL

```
 1: function SCHEDULER($NT$)
 2:   $W$.enqueue($NT$)
 3:   Remove all completed tasks from $R$
 4:   for task $\in$ $R, W$ do
 5:     task.xfactor $\leftarrow$ UpdateXfactor(task)
 6:   end for
 7:   if $W$.isEmpty() then
 8:     Restart tasks in $R$ with higher $cc$ if needed
 9:   end if
10:   for task $=$ $W$.peek() do
11:     small = isSmall(task);  $CL_{src} = CL_{dst} = []$
12:     if !saturated(src) $\wedge$ !saturated(dst) $\vee$ small then
13:       Schedule task
14:     else
15:       if saturated(src) then
16:         $CL_{src} \leftarrow$ FindTasksToPreempt(src,task)
17:       end if
18:       if saturated(dst) then
19:         $CL_{dst} \leftarrow$ FindTasksToPreempt(dst,task)
20:       end if
21:       Preempt tasks in $CL_{src} \cup CL_{dst}$ and schedule task
22:     end if
23:   end for
24: end function
```

If there are no queued tasks for a scheduling cycle (i.e., W is empty), SEAL increases the concurrency of running tasks if there is unused bandwidth at both the source and destination for each task (Listing 1, lines 7 to 9). This scenario can arise when some running tasks complete, remaining running tasks cannot consume additional bandwidth at their current concurrency, and there are no waiting tasks. To maximize utilization in such scenarios, SEAL considers the running tasks (R) in the descending order of remaining bytes to transfer (additional bandwidth is assigned preferentially to larger tasks) and attempts to increase their concurrency if

Listing 2 UpdateXfactor, FindThrCC, FindTasksToPreempt.

```
25: function UPDATEXFACTOR(task)
26:   [idealCC, idealThr] $\leftarrow$ FindThrCC(task, true)
27:   [bestCC, bestThr] $\leftarrow$ FindThrCC(task, false)
28:   $TT_{ideal} = \frac{task.num\_bytes\_total}{idealThr}$
29:   $TT_{load} = \frac{task.num\_bytes\_left}{bestThr} + task.TT_{trans}$
30:   return $\frac{task.WT+TT_{load}}{TT_{ideal}}$
31: end function
32: function FINDTHRCC(task, forIdealThr)
33:   predThr=0;  cc=0;  dst_cc=src_cc=0
34:   if ! forIdealThr then
35:     dst_cc = dst.cc;  src_cc = src.cc
36:   end if
37:   do
38:     bestThr = predThr; cc++
39:     predThr$\leftarrow$ throughput(src, dst, cc, src_cc, dst_cc, size)
40:   while ($predThr > bestThr \times \beta$) $\wedge$ ($cc < maxCC$)
41:   return [cc, bestThr]
42: end function
43: function FINDTASKSTOPREEMPT(host, task)
44:   $R_{host}$ = Tasks $\in R$ associated with host; $CL = []$
45:   do
46:     rtask = $R_{host}$.peek()
47:     if rtask.xfactor $>$ pf$\times$ task.xfactor then
48:       $CL$.append(rtask)
49:     end if
50:     xfactor$'$ $\leftarrow$ UpdateXfactor(task) s.t. $R = R - CL$
51:   while (xfactor$'$ $>$ $R$.last().xfactor) $\wedge$ (rtask $\neq$ $R_{host}$.last())
52:   return $CL$
53: end function
```

their source and destination are not saturated (saturation is determined using the procedure described in paragraph 4 in §4.3).

If W is not empty, then we consider two scenarios for each task in W. If neither the task's source nor destination is saturated, or the task is small (<1GB), then we schedule it with appropriate concurrency (Listing 1, lines 12, 13). A task's concurrency is determined by using the FindThrCC function in Listing 2. Specifically, concurrency starts at 1 and is increased only if the new predicted throughput is higher than the old by a user-defined threshold (e.g., 10%, chosen empirically). A lower threshold results in each task getting a higher concurrency and reduces the number of tasks that can run concurrently, since every endpoint has a limit (set by system administrators) on total concurrent transfers. However, a higher threshold also reduces the bandwidth that a single task can consume, leading to suboptimal bandwidth utilization under low-load conditions.

Alternatively, the waiting task's source and/or destination is saturated. We then consider preempting tasks associated with the waiting task's source and/or destination. We evaluate each such task in R, and add it to the candidate list of tasks (CL) for preemption if its *xfactor* is lower than that of the waiting task by *pf* or more (*pf* is the *preemption factor* described in §4.3. We tested several *pf* values and determined that SEAL is not overly sensitive to this value: it works well if *pf* is neither too close to 1.0, which causes too many preemptions, nor too large, which results in no preemptions. The results in §6.3 are with *pf*=1.5.) As each candidate task is added to CL, the waiting task's *xfactor* is recalculated but using a version of R that does not include the tasks in CL. If the new *xfactor* is sufficiently low, there are enough tasks in CL. Otherwise, we repeat until either CL is large enough or no more tasks are available for preemption. This case is covered in Listing 1, lines 15 to 21, and FindTasksToPrempt function in Listing 2.

We note that if average load is less than 100%, then all transfers will be scheduled, so that there is no possibility of starvation. Also, our experiments show that SEAL reduces maximum slowdown relative to baseline algorithms for all traces that we consider.

4.5 Preemptive Transfers

The state required to preempt/restart a transfer task is just the file offset (or, when transferring chunks of a file in parallel, a set of start, end offsets for missing blocks). As long as the transfer mechanism supports checkpointing this information and allows partial file transfers, one can preempt and restart a transfer with low overhead. Most advanced data transfer tools, including the Globus GridFTP used in our experiments, support these capabilities. The default checkpointing interval for Globus GridFTP is 5 secs. In the studies reported here, we set this interval to 1 sec, a value chosen empirically to balance unnecessary retransmission for preempted transfers against checkpointing overhead. We found that a checkpointing interval of 1 sec shows no statistically significant additional overhead relative to 5 sec, but reduces preemption retransmission overhead by 5x. (The biggest overhead incurred for a preemption is the retransmission of data moved in the maximum 1 sec and average 0.5 sec between the last checkpoint and preemption.)

Each transfer incurs a startup cost. Suppose that during a scheduling cycle the scheduler determines that task A, with an average throughput of T units per second, must be preempted and replaced by task B of higher priority. If task A is suspended immediately when this decision is made, then during the x secs that B takes to start, $T \times x$ throughput will likely be unused between source and destination. We avoid this waste by continuing to execute A until the first checkpoint information is received, indicating that B has started transferring data.

5. TYPE-AWARE SCHEDULING

We next formulate the load- and type-aware scheduling problem and describe our approach and the STEAL algorithm.

5.1 Batch and Interactive Transfers

The classification of transfers into *batch* and *interactive* provides an opportunity to schedule transfers adaptively such that *batch* transfers have minimal impact on *interactive* transfer performance, but use as much spare bandwidth as possible. This creates a bi-objective optimization problem, with the two metrics stated earlier in §3.2.2.

In §4.1, we defined a file transfer request by a six-tuple. We now add a boolean that indicates whether or not the transfer request is *batch*, to obtain a seven-tuple: <*source host*, *source file path*, *destination host*, *destination file path*, *file size*, *arrival time*, *batch*>. All assumptions made in §4.1 apply here: i.e., transfer requests arrive online, resources involved have different capabilities, and load on each resource can vary over time. In addition, we assume that transfer requests with *batch* set to *true* want to use only the bandwidth unused by *interactive* tasks and do not have any time constraints. To avoid indefinite delay, a *batch* task can be set to switch to *interactive* after a certain amount of time.

5.2 The STEAL Algorithm

We update the SEAL algorithm as shown in Listings 3 and 4 to define the STEAL algorithm, which schedules based on transfer type. STEAL gives lower priority to *batch* tasks so that they use only the bandwidth left after scheduling all *interactive* tasks. We still prioritize *batch* tasks by *xfactor*, but multiply their *xfactor* values by a small fraction to lower their priority relative to *interactive* tasks (Listing 3, lines 29.1 to 29.3). To maximize the use of excess bandwidth by *batch* tasks, STEAL uses appropriate transfer concurrency and eliminates preemption of *batch* tasks by other *batch* tasks. Thus, a waiting *batch* task *B2* cannot directly preempt a running *batch* task *B1* even when *B2*'s priority becomes higher than *B1*. (Assuming no other spare capacity, *B2*'s next chance to run will thus be when *B1* either completes or is pre-empted by an *interactive* task.)

Listing 3 Updates to the pseudocode presented in Listings 1 and 2.

```
      function Scheduler
7:    if $W$.isEmpty() $\vee$ $W$.InteractiveTasks.isEmpty() then
8:        Restart $R$.InteractiveTasks with higher $cc$ if needed
9:    end if
9.1:  if $W$.isEmpty() then
9.2:      Restart $R$.BatchTasks with higher $cc$ if needed
9.3:  end if
10.1: if isBatch(task) then
10.2:     if !saturated'($\lambda$, src) $\wedge$ !saturated'($\lambda$, dst) then
10.3:         Schedule task
10.4:     end if
10.5:     continue
10.6: else
10.7:     $B$ = Batch tasks in $R$ associated with src and/or dst
10.8:     $CL \leftarrow$ FindBatchTasksToPreempt(task, $B$)
10.9:     if !$CL$.isEmpty() then
10.10:        Preempt all tasks in $CL$ and schedule task; continue
10.11:    end if
10.12: end if

      function UpdateXfactor
29.1: if isBatch($task$) then
29.2:     return $0.00001 \times \frac{task.WT+TT_{load}}{TT_{ideal}}$
29.3: end if
```

Listing 4 FindBatchTasksToPreempt function.

```
54: function FindBatchTasksToPreempt(task, $B$)
55:     goalThr $\leftarrow$ FindThrCC(task, false)[1] s.t. $R = R - B$
56:     thr $\leftarrow$ FindThrCC(task, false)[1]; $CL = []$
57:     do
58:         btask = $B$.peek(); $T = R - CL - btask$
59:         thr' $\leftarrow$ FindThrCC(task, false)[1] s.t. $R = T$
60:         if thr' > thr then
61:             thr = thr'; $CL$.append(btask)
62:         end if
63:     while (thr < goalThr) $\wedge$ (btask $\neq$ $B$.last())
64:     return $CL$
65: end function
```

Referring back to the Listings 3 and 4, there are three scenarios.

Scenario 1 (W is empty): STEAL first works to increase the concurrency of running *interactive* tasks, with a preference for larger tasks. If unused bandwidth remains and is more than that a factor 1-λ times the maximum achievable bandwidth ($0 \leq \lambda \leq 1$, is a user-defined fraction to limit bandwidth for *batch* tasks) for both source and destination, STEAL attempts to increase the concurrency of running *batch* tasks to utilize the unused bandwidth. This is shown in Listing 3, lines 7 to 9.3.

Scenario 2 (W is not empty; contains only *batch* tasks): STEAL first attempts to increase the concurrency of running *interactive* tasks (preferentially towards larger tasks): see Listing 3, lines 7 to 9. If unused bandwidth remains and is more than 1-λ times the maximum achievable bandwidth for both source and destination, STEAL attempts to schedule waiting *batch* tasks to utilize the unused bandwidth (Listing 3, lines 10.1 to 10.5). The function **saturated'** determines whether an endpoint is saturated as described in paragraph 4 in §4.3 but by using λ to determine saturation limits.

Scenario 3 (W is not empty; contains at least one *interactive* task; may or may not contain *batch* tasks): In the case of an *interactive* task, there are two cases. In the first case, one or more running *batch* tasks are associated with the source and/or the destination of the waiting *interactive* task under consideration. STEAL then attempts to preempt one or more of these *batch* tasks to free up the amount of bandwidth that the model predicts that the given *in-*

teractive task would obtain in their absence. Note that this step does not necessarily preempt all *batch* tasks associated with the source and the destination, as the *interactive* task may not be able to use all available bandwidth at both source and destination. Details are shown in Listing 3, lines 10.7 to 10.11 and Listing 4. In the second case, no *batch* tasks are running. `STEAL` then behaves in the same way as `SEAL`: see the two scenarios in paragraphs 3 and 4 in §4.4. If there are *batch* tasks in W, they will be considered only after all *interactive* tasks in W are scheduled, at which point the scenario is the same as 'Scenario 2' above.

6. EXPERIMENTAL EVALUATION

We evaluated `SEAL` and `STEAL` with real traces in a wide-area environment. We use GridFTP [2], which extends the legacy File Transfer Protocol to enable secure, reliable, and fast transport of bulk data, as the underlying data transfer protocol for our experiments. Various GridFTP implementations [3, 19, 26] are widely used for high-speed data movement. We used the Globus implementation of GridFTP for our experiments. Although we evaluated our algorithms in the context of GridFTP, the scheduling problem that we consider is more general and our algorithms are generally applicable for wide-area file transfers.

To achieve concurrency, we exploit the partial file transfer support (i.e., transfer *X* bytes of data from offset *Y*) found in Globus GridFTP. In order to avoid additional overhead, we ensure that partial transfer sizes are set to be larger than the *bandwidth-delay product* for the given network link. It should be noted that GridFTP [3] (as well as other tools such as BBCP [5]) can also split a file among multiple TCP streams in order to accelerate transfers. However, this method typically does not involve any parallelism at the storage I/O level. In comparison, multiple independent transfers that we use, with each transferring a partial file, achieves parallelism at both network and storage levels.

We first describe the environment and traces, and then present results obtained when comparing `SEAL` with *baseline* algorithms and when *batch* tasks are added and the `STEAL` algorithm is applied.

6.1 Environment

In order to simplify access and allow repeated experiments, we conducted experiments in an environment involving one source and five destinations. Analysis of GridFTP transfer logs, network requirement reports from science communities [20], and cyberinfrastructure usage reports [55] suggest that a large fraction of bandwidth consumption at many large facilities involves data transfers to/from large/medium-scale facilities. Thus, this scenario captures an important scheduling use case.

Our source is a DTN at Stampede, a supercomputer at the Texas Advanced Computing Center. Destinations are DTNs at five other supercomputing centers: the Blacklight supercomputer at the Pittsburgh Supercomputing Center; Darter at the University of Tennessee; Gordon at the San Diego Supercomputer Center; Mason at Indiana University; and Yellowstone at National Center for Atmospheric Research. The various DTNs are dedicated for file transfer and each has 10Gbps WAN connectivity. The DTN at Stampede can achieve >9Gbps aggregate disk-to-disk throughput; Yellowstone, Gordon, Blacklight, Mason, and Darter can achieve $\sim$8Gbps, 7Gbps, 4Gbps, 2.5Gbps, and 2Gbps, respectively. Since these resources are production DTNs, we ran our experiments at night and weekends so as not to disrupt production activities. Unless otherwise noted, each result is an average of at least three runs.

6.2 Workload (Traces) Used for Evaluation

In order to allow repeatable experiments, we used real traces as workloads. We obtained these traces from the anonymized usage statistics that Globus GridFTP servers send to a usage collector. These logs include transfer size, start time, and end time. Since our goal is to handle situations in which resources are not (vastly) over-provisioned, we obtained our traces by first selecting the 10 servers that transferred most bytes in a one month period, then picking the day in that month in which the most bytes were transferred by those servers, and finally using the log from the one server among those 10 that transferred the most bytes on that day.

Table 2: File size distribution and total number of file transfer tasks for each of the four traces used in our experiments

Log	**0-1M**	**1-10M**	**10-100M**	**100M-1G**	**1-10G**	**All**
25%	247	40	72	46	117	522
45%	907	168	334	209	232	1850
60%	728	11	34	318	290	1381
45%-4x	117	125	19	109	74	444

Since our execution environment is a production infrastructure in continuous use, we were limited in the length of our experiments. Thus, we selected from the chosen 24-hour log three 15-minute traces with different loads: the `25%`, `45%`, and `60%` traces, respectively. Average load of the 24-hour workload was $\sim$25%. We looked at all non-overlapping 15-minute windows in the 24-hour period and picked one with the same average load as the entire workload (25%). The coefficient of variation of 1-minute average concurrent transfers is approximately the same, too. We picked one that had the highest load ($\sim$60%), and one with $\sim$45% load (which is in between 25% and 60%). We define *load* as the total volume of all file transfers in the 15-minute trace, divided by the maximum amount of data that the source can transfer in a 15-minute period. Since the maximum achievable throughout for Stampede is 9.2Gbps, the maximum that it can transfer in 15 minutes is $\sim$1TB. Thus, the total volume of the transfers in the `25%`, `45%`, and `60%` traces are $\sim$250GB, 450GB, and 600GB, respectively.

For our experiments, we replay the transfers recorded in these traces. The Globus usage collector does not record destination identifiers; to address this problem, we randomly split transfers among the five destinations, with weighted probability based on their capacities.

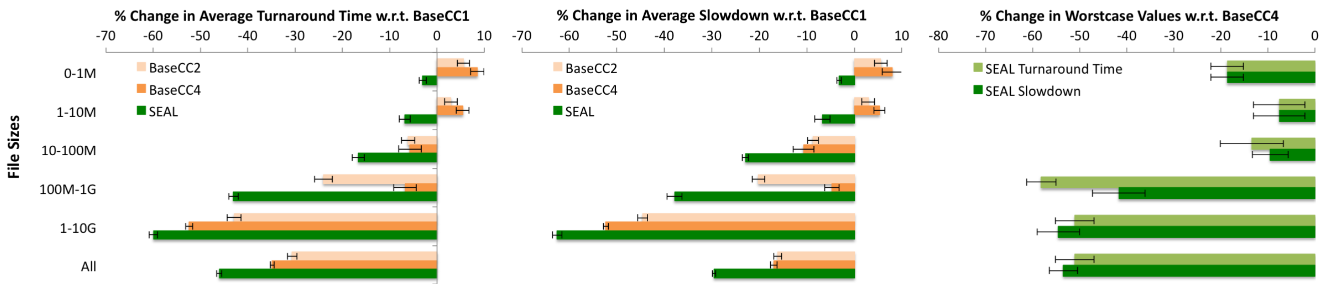
Table 2 shows the number of file transfers and file size distribution for each trace. We use five categories: transfers with file size, f: $\leq$1MB (`0-1M`); 1MB$>$ f $\leq$10MB (`1-10M`); 10MB$<$ f $\leq$100MB (`10-100M`); 100MB$<$ f $\leq$1GB (`100M-1G`); and 1GB$<$ f $\leq$ 10GB (`1-10G`). There are no files $>$10GB in these traces. We see that the traces are heavily weighted toward small and medium files. This situation may change in the future, as science communities move to larger and fewer files [20]. To capture some of these trends, we created another `45%-4x` trace by starting with the `45%` trace, selecting 25% of all files randomly, and replacing each of the selected files with a transfer four times larger.

6.3 Evaluation of SEAL

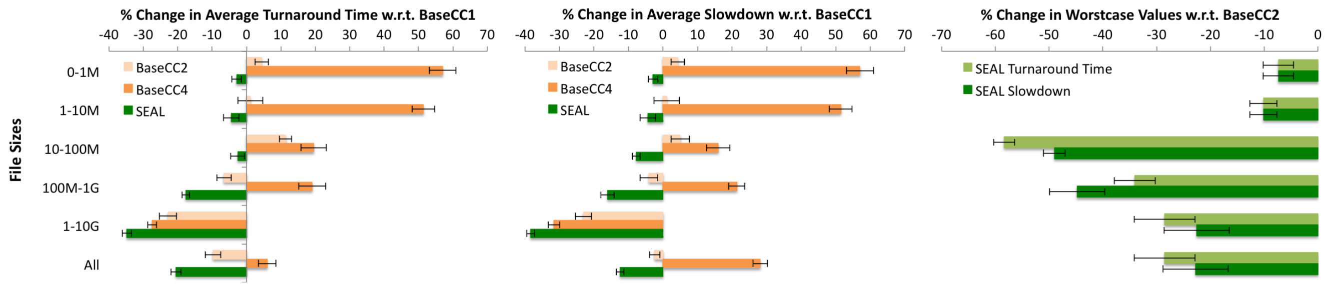
We compare the performance of `SEAL` with that of three baseline algorithms. The first, `BaseCC1`, replicates the common current practice of scheduling each file transfer as it is submitted and using only parallel TCP streams to improve performance. Specifically, we use 2 parallel TCP streams as that was the most widely used value in our logs; in addition, additional TCP streams did not provide any significant performance improvement in our experimental environment, in the absence of storage-level parallelism.

The other two baseline versions, `BaseCC2` and `BaseCC4`, use both parallel storage operations and parallel TCP streams for individual transfers, with static per-file concurrency settings of 2 and 4, respectively. (The performance of the baseline scheme with per-file concurrency values $>$4 was worse than that of `BaseCC2` and `BaseCC4`.) For $\leq$10MB files, we use a concurrency (CC) of 1 for all algorithms, since splitting such small files can only hurt performance. (This threshold was computed based on the bandwidth-delay product of the source-destination links in our experimental environment.)

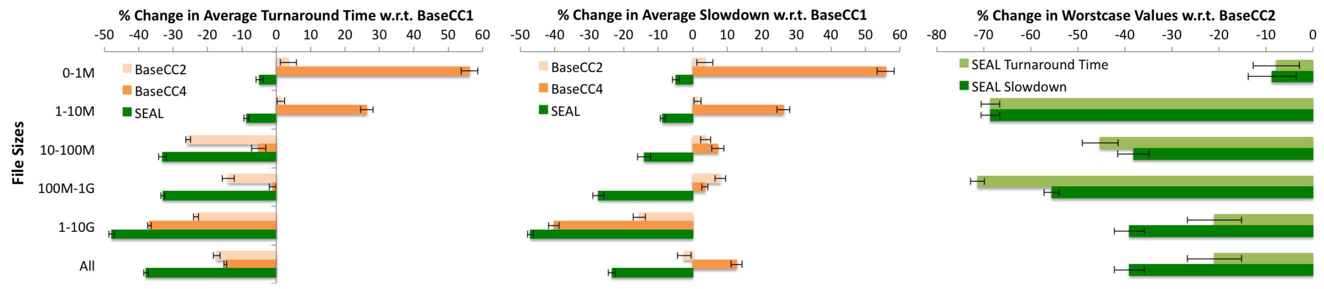
We focus on two metrics: average slowdown (a ratio) and aver-



(a) Results for `25%` trace



(b) Results for `45%` trace



(c) Results for `60%` trace

Figure 3: Percentage change in average turnaround time (left) and slowdown (center) relative to `BaseCC1` for various algorithms; and (right) percentage change in worst case turnaround time and slowdown for `SEAL` relative to best performing baseline. Negative values are better.

age turnaround time (in seconds). We report the percentage change in these metrics for `BaseCC2`, `BaseCC4`, and `SEAL` relative to `BaseCC1`. Specifically, we report the observed percentage change in these metrics for the different file size classes. We also report the percentage change in worst-case slowdowns and turnaround times for `SEAL` relative to the best-performing baseline scheme, which is either `BaseCC2` or `BaseCC4` depending on the trace.

Figures 3a–3c show results for the `25%`, `45%`, and `60%` traces, respectively. In each figure, the leftmost chart shows the percentage change in average turnaround times for `SEAL` and the baseline algorithms `BaseCC2` and `BaseCC4` relative to the baseline algorithm `BaseCC1`; the center chart shows the corresponding percentage changes in average slowdown; and the rightmost chart shows the percentage changes in worst-case slowdowns and turnaround times for `SEAL` compared to the best baseline algorithm. More negative values are better in each case.

We see that `SEAL` performs better, in terms of both average slowdown and turnaround time, than all baseline algorithms, both overall and for each individual file size category and trace. `SEAL` even performs better for $\leq$10MB tasks, for which all algorithms use the same parameters, because `SEAL`'s load awareness postpones larger tasks when load is high. Also, blindly using higher CC for >10MB tasks (along with scheduling all tasks upon arrival) hurts $\leq$10MB tasks (`BaseCC4` is almost always worst there). Even though the priority for larger tasks (`1-10G`) grows relatively slowly in `SEAL`, those tasks still benefit from load-aware scheduling and dynamic adjustment of concurrency. `10-100M` and `100M-1G` tasks benefit from relatively higher prioritization than `1-10G` tasks, load-aware scheduling and dynamic scaling of concurrency. But the range of concurrency values for `10-100M` tasks is limited as splitting these relatively small files into too many pieces can hurt the performance. Thus, `100M-1G` tasks get the most benefit from `SEAL`.

Averaged over all task categories, `SEAL` outperforms the best baseline algorithm in average slowdowns by 16%, 11%, and 22%, for the three traces, respectively; and in average turnaround times by 18%, 12%, and 25%. In terms of worst-case slowdown and turnaround time, `SEAL` is significantly better than the best baseline algorithm for most task categories; it reduces the worst-case slowdowns and turnaround times by as much as 50% for some categories in `25%` and `45%` traces and by as much as 70% for some categories in `60%` trace. The factors that help `SEAL` perform consistently better include load awareness, prioritization of tasks based on their slowdowns, and sufficient but not excessive concurrency.

The baseline algorithms respond differently to changes in trace, metric, and task category. For the lowest-load trace, `BaseCC4` outperforms `BaseCC2` (on average across all transfers); `BaseCC2` is better when load increases. As another example, for `100M-1G` tasks in the `60%` trace, `BaseCC2` has a lower average turnaround time, whereas `BaseCC1` has a lower average slowdown. `SEAL` outperforms the best baseline algorithm for both metrics, both in terms of the overall average and in terms of the average for individual task categories for all three traces. These results show that

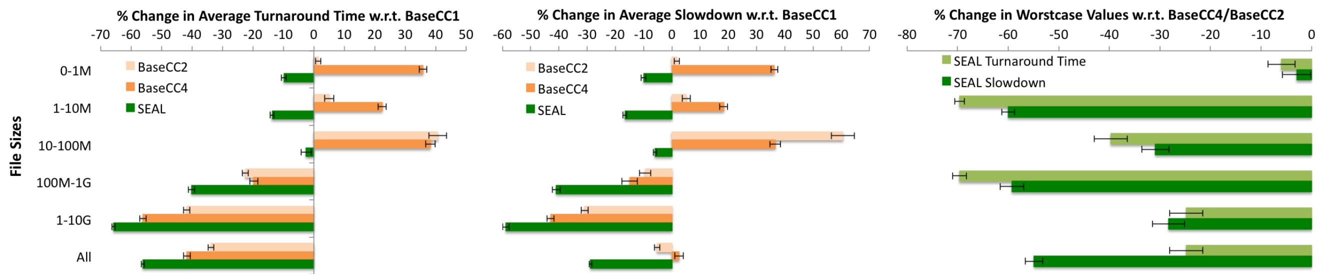


Figure 4: Results for `45%-4x` Trace. Percentage change in average turnaround time and slowdown compared to `BaseCC1` for various algorithms; and percentage change in worstcase turnaround time and slowdown for `SEAL` compared to best performing baseline.

`SEAL` performs well irrespective of trace characteristics.

Across the three traces, `SEAL` achieves the least improvement in average slowdown and turnaround time for `45%`, relative to the best baseline algorithm. This trace had the most (1850) files. Since our experiments were run on production environments, we limited the total concurrency at any time to 80 (the maximum at most sites is 100) to avoid disrupting production activities. With many files, we hit this total concurrency cap more often and had less flexibility in increasing concurrency for big files. Recalling the motivation for and description of the `45%-4x` trace in §6.2, we do not expect this limitation to arise with this trace, and indeed we see in Figure 4 that for this trace, `SEAL` again outperforms the other algorithms, particularly for average slowdown. Specifically, `SEAL` improves over `BaseCC2` by only 11% for `45%`, but by 25% for `45%-4x`. This result indicates that `SEAL` provides significant benefit over the best baseline algorithm if it has reasonable flexibility in the concurrency values to use for tasks—yet can still perform better than the best baseline even in constrained scenarios.

In a final set of `SEAL` experiments, we compared with a yet more sophisticated baseline algorithm. The motivation for this study was as follows. If one considers the results of just one trace, for example `60%`, one might argue that a baseline algorithm that simply uses a different concurrency based on file size might perform better than or as well as `SEAL`. As the center chart in Figure 3c shows, the best baseline for files $\leq$1GB is `BaseCC1` and for files $>$1GB is `BaseCC4`. Therefore, we defined a new baseline `BaseVary` algorithm that uses a concurrency of 1 for files $\leq$1GB and a concurrency of 4 for files $>$1GB. Figure 5 shows that `SEAL` also outperforms `BaseVary` for all task categories: by $\geq$20% overall and $\geq$40% for `100M-1G`. This result shows that simply using a different concurrency based on file size provides little benefit.

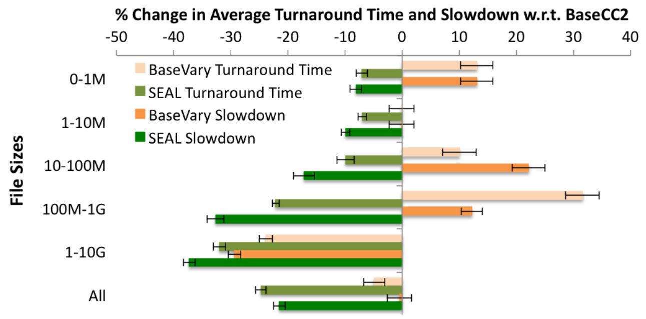


Figure 5: Percentage change in average turnaround time and slowdown for `BaseVary` and `SEAL` compared to `BaseCC2` for the `60%` trace.

6.4 Evaluation of STEAL

We use four traces to evaluate `STEAL`: the `25%`, `45%`, and `60%` traces described in §6.2 plus a 60% high variation (`60%-HV`) trace with greater variation in the load due to *interactive* tasks. The one-minute-averaged concurrent transfer counts in the `60%` and `60%-HV` traces have standard deviations of 17.31 and 48.56, respectively. (`60%-HV` is from the same GridFTP server as the others, but for a different 24-hour period.)

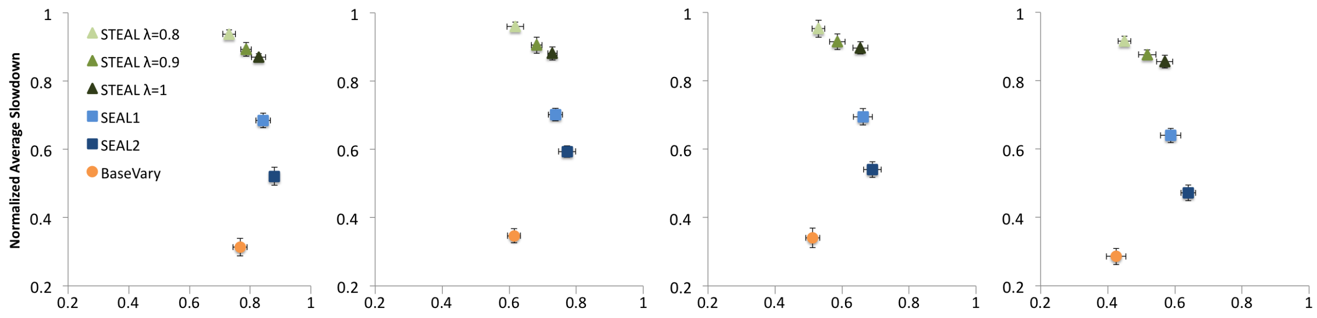
For each trace, we then defined the original tasks to be *interactive* and added enough 50GB *batch* tasks to consume the bandwidth unused by *interactive* tasks over the 15-minute duration. (By setting the aggregate *batch* task size to match the unused bandwidth, we simplify comparison of the different schemes, as even the `BaseVary` and `SEAL` schemes, which do not differentiate *batch* and *interactive*, can complete the *interactive* tasks during the 15-minute experimental window.) We make the *batch* tasks available to be scheduled right from the beginning of the 15-minute period, and split them among the five destinations based on their capacities. As noted in §6.2, the maximum data that can be transferred between the source and destinations in our experimental environment in 15 minutes is ∼1TB, and the interactive load for `25%`, `45%`, `60%`, and `60%-HV` traces are ∼250GB, 450GB, 600GB and 600GB, respectively. Thus, we used 17×50GB=850GB, 13(650GB), 10(500GB), and 10(500GB) *batch* jobs, respectively, for the four traces.

We evaluate `STEAL` for λ = {1, 0.9, 0.8}, indicating that *batch* tasks can use up to {100%, 90%, 80%}, respectively, of the bandwidth unused by interactive jobs. We also evaluate `BaseVary` and two `SEAL` variants, `SEAL1` and `SEAL2`, with the following motivation for the latter. Because `SEAL` does not distinguish between *interactive* and *batch* tasks, the larger size of the *batch* tasks in our experiments means that their priorities (*xfactor*s) will increase at a slower rate than that of *interactive* tasks (for same or similar endpoints) that arrive at the same time. Therefore, even under `SEAL`, *batch* jobs may often be preempted and/or queued until their wait time becomes high. Given that we cannot perform long experiments, we thus define `SEAL1` and `SEAL2` as follows. In `SEAL1`, batch tasks arrive at the start of the schedule for each 15-minute trace. In `SEAL2`, we increase *xfactors* as if the *batch* tasks had arrived an hour before the start of the schedule for 15-minute traces under consideration. Thus *interactive* tasks have an advantage in `SEAL1` and *batch* tasks have an advantage in `SEAL2`.

As noted in §3.2.2 and §5.1, we have a bi-objective problem—minimizing the average slowdown of interactive transfers and maximizing the utilization of unused bandwidth for *batch* transfers. Both metrics introduced in §3.2.2 take a value between 0 and 1, and a value close to 1 is desirable in each case. Also, in calculating the *normalized slowdown* ($\frac{SD_I}{SD_{I+B}}$), the average slowdown for interactive jobs, SD_I, is obtained by executing interactive jobs alone under `SEAL`.

Figure 6 shows that `STEAL` (for different λ values) performs significantly better for *interactive* tasks, in terms of slowdown (y-axis), than `SEAL` and `BaseVary`, as it explicitly prioritizes *interactive* over *batch*. `STEAL/`λ`=1` is also better than `BaseVary`, and comparable to `SEAL1` and `SEAL2`, in its use of spare bandwidth for *batch* tasks (x-axis). Note that the best performance is in the upper-right corner in these graphs.

We attribute these good results to `STEAL`'s reduction of interfer-



(a) `25%` trace (b) `45%` trace (c) `60%` trace (d) `60%-HV` trace

Figure 6: Results when both minimizing average slowdown for *interactive* tasks and maximizing spare bandwidth utilization for *batch* tasks. X-axis: 'fraction of spare bandwidth used' by *batch* tasks.

ence between *batch* and *interactive* tasks, maximum exploitation of periods of no/low *interactive* load by saving *batch* tasks for those periods, and elimination of *batch* tasks preempting one another. Although `SEAL2` performs the best in its use of spare bandwidth for *batch* tasks, it does so with a significant negative impact on *interactive* performance. `STEAL` uses less spare bandwidth for `60%-HV` than for `60%` because `60%-HV` trace's high variation in *interactive* load increases preemption of *batch* tasks. But the performance of *interactive* tasks for `60%-HV` is quite close to that for `60%`.

With `STEAL`, lower λ values result in less bandwidth for *batch* tasks. Thus, decreasing λ improves the performance of *interactive* tasks—meaning that *batch* tasks (being larger) are stealing bandwidth from *interactive* tasks. Although `STEAL` gives *batch* tasks lower priority, those *batch* tasks that get scheduled still get the best concurrency value (a relative higher value since *batch* tasks are larger) for that task. System operators can tune λ to trade off *interactive* task performance and system utilization.

We also see in Figure 6 that the fraction of the spare bandwidth used by `BaseVary` is best for `25%` and gets worse as the *interactive* load increases. This result is expected because `25%` has many *batch* tasks and thus `BaseVary` has a high aggregate concurrency value throughout the schedule, resulting in higher utilization. However, the high aggregate concurrency value for *batch* tasks severely affects *interactive* performance, as the *batch* tasks consume 80% of the total concurrency allowed when they are running.

In summary, in the presence of *batch* tasks, `STEAL/`λ`=1` is 63%, 21%, and 39% better than `BaseVary`, `SEAL1`, and `SEAL2`, respectively, in terms of the average slowdown for *interactive* tasks, averaged across the four traces; the fraction of spare bandwidth used by *batch* tasks with `STEAL/`λ`=1` is 18% better, 1.75% worse, and 7.5% worse than `BaseVary`, `SEAL1`, and `SEAL2` respectively, averaged across the four traces. We also measured the average turnaround time for *interactive* tasks for all the schemes considered above and observed similar trends; we do not present those results here due to space constraints. We thus conclude that `STEAL`'s differentiation between *interactive* and *batch* tasks improves the response time of *interactive* tasks significantly with little negative impact on the bandwidth used by *batch* tasks as compared to `SEAL`.

7. RELATED WORK

Coffman et al. [17] established that scheduling file transfers in a distributed network to ensure minimum completion time is NP-complete. Many heuristics have since been proposed [28, 29]. However, because these algorithms require that transfer requests be known in advance, they normally cannot be applied in practice.

Content distribution networks (CDNs) [52] and BitTorrent [18] are widely used for high-speed distribution of/access to popular files. However, our experience, based for example on GridFTP data [5000+ servers move >1.5 PB (>25M files) per day on average, as of July, 2015] is that much science traffic involves different patterns: data are not widely replicated (as in BitTorrent) nor multicast (as in CDNs). BitTorrent employs incentive-based file sharing where peers that contribute more data at faster rates get preferential treatment for downloads. Our methods could also be applied to manage resources in CDNs and BitTorrent.

Foster et al. [27] use differentiated service mechanisms to schedule file transfers of differing priority, but do not consider concurrency. Algorithms have been developed that use multiple paths to improve transfer performance [14]. In the work reported here, transfers take default network paths. The Stork [39] data placement scheduler allows the use of directed acyclic graph schedulers to encapsulate dependencies between computation and data movement. Our algorithms can be incorporated into a system like Stork. Researchers have proposed differentiated treatment of traffic classes by using priority queues at routers or similar techniques [27, 32, 41, 42]. Our approach, in contrast, does not require router support, kernel modifications, or additional storage infrastructure.

Wolski [54], Vazhkudai et al. [51], and Lu et al. [44] proposed methods for predicting transfer performance. While we focus instead on scheduling wide-area file transfers, we can use some of these methods for throughput estimation in applicable scenarios.

8. CONCLUSIONS

We have defined two new algorithms for efficient online scheduling of wide-area file transfers. The first, `SEAL`, uses data-driven models of transfer performance to vary the concurrency of individual transfers, while also queuing and preempting tasks, subject to constraints on the aggregate concurrency for all transfers. By thus delaying some transfers, `SEAL` improves average slowdown for all transfers when load is high; by increasing concurrency for other transfers, it increases aggregate throughput when load is low. Our second algorithm, `STEAL`, uses user-supplied transfer types to further optimize schedules. `STEAL` treats *batch* and *interactive* transfers differently, allocating bandwidth unused by *interactive* transfers to *batch* transfers, while being largely non-intrusive to *interactive* transfers.

We evaluated our algorithms by using real traces and on a production system. We showed significant improvements over the state of the art in terms of average and worst-case slowdowns as well as turnaround times. We also showed that `STEAL` can allow *batch* tasks to successfully use a large portion of the excess bandwidth, while being significantly better than `SEAL` in terms of sustaining the performance of *interactive* tasks.

9. ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357 and Office of Advanced Scientific Computing Research under the Next Generation Networking for Science Program funded RAMSES project, and the National Science Foundation, under grants ACI-1339757, ACI-1339798 and ACI-1440761.

10. REFERENCES

[1] K. Aida, H. Kasahara, and S. Narita. Job scheduling scheme for pure space sharing among rigid jobs. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98, pages 98–121, London, UK, 1998. Springer-Verlag.

[2] W. Allcock. GridFTP protocol specification (Global Grid Forum recommendation GFD.20), 2003.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 54–, Washington, DC, USA, 2005. IEEE Computer Society.

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

[5] BaBar Copy. http://slac.stanford.edu/~abh/bbcp/.

[6] G. Bell and M. Ernst. HEP Community Summer Study 2013 Computing Frontier: Networking. In *Snowmass'2013*.

[7] Belle-II Experiment Network Requirements, Oct. 2012. http://www.osti.gov/scitech/servlets/purl/1171367.

[8] Biological and Environmental Research Network Requirements, Nov. 2012. http://www.es.net/assets/pubs_presos/BER-Net-Req-Review-2012-Final-Report.pdf.

[9] K. Bergman, V. Chan, D. Kilper, I. Monga, G. Porter, and K. Rauschenbach. Scaling terabit networks: Breaking through capacity barriers and lowering cost with new architectures and technologies. *NSF Workshop*, 2013.

[10] User facilities of the Office of Basic Energy Sciences, 2009. http://science.energy.gov/~/media/bes/suf/pdf/BES_Facilities.pdf.

[11] Basic Energy Sciences Network Requirements Workshop, September 2010 - Final Report. http://www.es.net/assets/Uploads/BES-Net-Req-Workshop-2010-Final-Report.pdf.

[12] Office of Science and Technology Policy, Executive Office of the President. Fact Sheet: Big Data Across the Federal Government. Washington, D.C., Mar. 29, 2012. https://www.whitehouse.gov/sites/default/files/microsites/ostp/big_data_fact_sheet_final.pdf.

[13] TCP backing off due to burstiness. http://fasterdata.es.net/assets/fasterdata/Using-tc-with-OSCARS-curcuits.pdf.

[14] Z. Cai, V. Kumar, and K. Schwan. IQ-Paths: Self-regulating data streams across network overlays. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, 2006.

[15] C. Castillo, G. Rouskas, and K. Harfoush. On the design of online scheduling algorithms for advance reservations and QoS in Grids. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–10, March 2007.

[16] A. Clematis, A. Corana, D. D'Agostino, A. Galizia, and A. Quarati. Job-resource matchmaking on Grid through two-level benchmarking. *Future Gener. Comput. Syst.*, 26(8):1165–1179, Oct. 2010.

[17] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and A. S. LaPaugh. Scheduling file transfers in a distributed network. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 254–266, New York, NY, USA, 1983. ACM.

[18] B. Cohen. Incentives build robustness in bittorrent. In *1st International Workshop on Economics of P2P Systems*, June 2003.

[19] dcache GridFTP. http://trac.dcache.org/wiki/gridftp.

[20] DOE data crosscutting requirements review, 2013. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/ASCR_DataCrosscutting2_8_28_13.pdf.

[21] DOE High Performance Computing Operational Review, 2014. http://www.osti.gov/scitech/servlets/purl/1163236.

[22] Data Transfer Nodes. http://fasterdata.es.net/science-dmz/DTN/.

[23] ESnet Strategic Plan. http://www.es.net/assets/Uploads/ESnet-Strategic-Plan-March-2-2013.pdf.

[24] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling – a status report. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'04, pages 1–16, Berlin, Heidelberg, 2005. Springer-Verlag.

[25] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS '97, pages 1–34, London, UK, 1997. Springer-Verlag.

[26] J. Feng, L. Cui, G. Wasson, and M. Humphrey. Toward seamless Grid data access: Design and implementation of GridFTP on .net. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 164–171, Washington, DC, USA, 2005. IEEE Computer Society.

[27] I. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. *Comput. Commun.*, 27(14):1375–1388, Sept. 2004.

[28] A. Giersch, Y. Robert, and F. Vivien. Scheduling tasks sharing files from distributed repositories. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 246–253. Springer Berlin Heidelberg, 2004.

[29] A. Goel, M. R. Henzinger, S. Plotkin, and E. Tardos. Scheduling data transfers in a network and the set scheduling problem. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 189–197, New York, NY, USA, 1999. ACM.

[30] GridFTP Usage Stats Collection. http://toolkit.globus.org/toolkit/docs/6.0/gridftp/admin/#gridftp-usage.

[31] Y. Gu and R. L. Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Comput. Netw.*, 51(7):1777–1799, May 2007.

[32] C. Guok, D. Robertson, M. Thompson, J. Lee, B. Tierney, and W. Johnston. Intra and Interdomain Circuit Provisioning Using the OSCARS Reservation System. In *3rd International Conference on Broadband Communications, Networks, and Systems*, 2006.

[33] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, N. Frontiere, and Z. Lukić. The universe at extreme scale: Multi-petaflop sky simulation on the bg/q. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 4:1–4:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[34] High Energy & Nuclear Physics Net. Req. Review, Aug. 2013. http://www.es.net/assets/Papers-and-Publications/HEP-NP-Net-Req-2013-Final-Report.pdf.

[35] W. E. Johnston, E. Dart, M. Ernst, and B. Tierney. Enabling high throughput in widely distributed data management and analysis systems: Lessons from the LHC. In *TERENA Networking Conference (TNC)*, 2013.

[36] S. D. Kahn. On the future of genomic data. *Science*, 331(6018):728–729, 2011.

[37] R. Kettimuthu et al. Lessons learned from moving earth system grid data sets over a 20 gbps wide-area network. In *19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 316–319, New York, NY, USA, 2010. ACM.

[38] R. Kettimuthu, G. Vardoyan, G. Agrawal, and P. Sadayappan. Modeling and optimizing large-scale wide-area data transfers. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0:196–205, 2014.

[39] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the Grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, pages 342–349, Washington, DC, USA, 2004. IEEE Computer Society.

[40] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour. On the design and evaluation of job scheduling algorithms. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '99/JSSPP '99, pages 17–42, London, UK, 1999. Springer-Verlag.

[41] A. Kuzmanovic and E. W. Knightly. TCP-LP: Low-priority service via end-point congestion control. *IEEE/ACM Trans. Netw.*, 2006.

[42] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 74–85, New York, NY, USA, 2011. ACM.

[43] LIGO Data Replicator. http://www.lsc-group.phys.uwm.edu/LDR/.

[44] D. Lu, Y. Qiao, P. Dinda, and F. Bustamante. Characterizing and predicting TCP throughput on the wide area network. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 414–424, June 2005.

[45] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 354–367, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[46] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, June 2001.

[47] Nuclear physics network requirements workshop, 2008. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Np_net_req_workshop.pdf.

[48] L. Ramakrishnan, C. Guok, K. Jackson, E. Kissel, D. M. Swany, and D. Agarwal. On-demand overlay networks for large scientific data transfers. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 359–367, Washington, DC, USA, 2010. IEEE Computer Society.

[49] Y. Ren, T. Li, D. Yu, S. Jin, T. Robertazzi, B. L. Tierney, and E. Pouyoul. Protocols for wide-area data-intensive applications: Design and performance issues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 34:1–34:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[50] Y. Sun, S. Marru, and B. Plale. Experience with bursty workflow-driven workloads in LEAD science gateway. In *Proceedings of TeraGrid Conference*, 2008.

[51] S. Vazhkudai and J. Schopf. Using regression techniques to predict large data transfers. *Int. J. High Perf. Comp. Appl.*, 2003.

[52] L. Wang, K. S. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the codeen content distribution network. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 14–14, Berkeley, CA, USA, 2004. USENIX Association.

[53] D. Williams et al. Earth System Grid: Enabling access to multi-model climate simulation data. *Bulletin of American Meteorological Society*, 90(2), 2009.

[54] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, Jan. 1998.

[55] XSEDE Report, 2013. http://ideals.illinois.edu/handle/2142/44872.

[56] E. Yildirim and T. Kosar. End-to-end data-flow parallelism for throughput optimization in high-speed networks. *Journal of Grid Computing*, 10(3):395–418, 2012.

[57] E. Yildirim, D. Yin, and T. Kosar. Prediction of optimal parallelism level in wide area data transfers. *IEEE Trans. Parallel Distrib. Syst.*, 22(12):2033–2045, Dec. 2011.