Joint Scheduling and Source Selection for Background Traffic in Erasure-Coded Storage

Shijing Li, Tian Lan, Moo-Ryong Ra, and Rajesh Panta

Abstract—Erasure-coded storage systems have gained considerable adoption recently since they can provide the same level of reliability with significantly lower storage overhead compared to replicated systems. However, background traffic of such systems – e.g., repair, rebalance, backup and recovery traffic – often has large volume and consumes significant network resources. Independently scheduling such tasks and selecting their sources can easily create interference among data flows, causing severe deadline violation. We show that the well-known heuristic scheduling algorithms fail to consider important constraints, thus resulting in unsatisfactory performance. In this paper, we claim that an optimal scheduling algorithm, which aims to maximize the number of background tasks completed before deadlines, must simultaneously consider task deadline, network topology, chunk placement, and time-varying resource availability. We first show that the corresponding optimization problem is NP-hard. Then we propose a novel algorithm, called Linear Programming for Selected Tasks (LPST) to maximize the number of successful tasks and improve overall utilization of the datacenter network. It jointly schedules tasks and selects their sources based on a notion of Remaining Time Flexibility, which measures the slackness of the starting time of a task. We evaluated the efficacy of our algorithm using extensive simulations and validate the results with experiments in a real cloud environment. Our results show that, under certain scenarios, LPST can perform 7x$\sim$10x better than the heuristics which blindly treat the infrastructure as a collection of homogeneous resources, and 21.7$\sim$65.9 percent better than the algorithms that only take the network topology into account.

Index Terms—Traffic scheduling, erasure code, storage

1 Introduction

Erasure-coding technology has been applied to many large scale storage systems. The technique allows us to significantly save storage space while still maintaining the same level of reliability as replicated systems [1], [2], [3], [4], [5], [6], [7], [8], [9]. In an (n,k) erasure code, a given data object or a file, is split into k pieces and encoded into n chunks ($n \geq k$), each stored on a different storage node[1] to maximize reliability. The file can be retrieved by querying any k-out-of-n chunks from these storage nodes, tolerating at most $n-k$ lost chunks. Compared with replicating data object $n-k$ times, (n,k) erasure-coding chunks save $(n-k+1)-\frac{k}{n}$ space. For example, $(9,6)$ erasure-coding save 83 percent space.

However, a major drawback of erasure-coding is that they generate large amounts of background traffic. The background traffic could be repair traffic generated when an erasure-coded chunk is lost, rebalance traffic when storage capacity is added or reduced, backup traffic, etc. As an example, repairing a data chunk, say x bytes of data, could generate kx bytes of network traffic in an (n,k) erasure-coded system. Background traffic has been shown to consume a substantial amount of datacenter network bandwidth. In [10], the authors characterized backup workloads in EMC Data Domain backup systems in production use and showed that on average, background traffic per week is equivalent to about 21 percent of total stored data. The vast majority of repair times are relatively short but had large deviation, leading to undesirable impact to the infrastructure [11]. Recent measurements on a Facebooks data warehouse cluster storing multiple petabytes of erasure-coded data, required a median of more than 180 Terabytes of data transferred to recover from 50 machine-unavailability events per day [12].

Existing systems often schedule each background task independently [13]. With reasonably high probability, these distributed tasks share same deadlines, compute, network and storage resources of the underlying infrastructure. These aspects cause interference among competing data flows, resulting in poor resource utilization and frequent violation of Service Level Agreements (SLAs) associated with those tasks.

To mitigate the problem, this paper proposes a novel and practical way of scheduling background jobs in a holistic manner, by jointly considering all background jobs together. Specifically, we solve the following problem—given a set of background tasks with known deadlines, how should the tasks be scheduled and the sources be selected, such that the number of tasks that successfully complete within their deadline is maximized? We consider a joint, online optimization of background traffic over task scheduling and source selection

1. Server, node, machine all refer to the same entity in this paper.

• *S. Li and T. Lan are with the ECE, George Washington University, Washington, DC 20052. E-mail: {shijing, tlan}@gwmail.gwu.edu.*
• *M.-R. Ra and R. Panta are with the AT&T Labs Research, Florham Park, NJ 07932. E-mail: {mra, rpanta}@research.att.com.*

Manuscript received 9 Oct. 2017; revised 23 May 2018; accepted 27 May 2018. Date of publication 11 June 2018; date of current version 9 Nov. 2018. (Corresponding author: Shijing Li.)
Recommended for acceptance by F. Qin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2845845

to maximize the number of tasks meeting deadlines in erasure-coded storage systems.

Deadline-aware scheduling has been studied extensively in many domains [14], [15], [16], [17], [18]. However, scheduling background tasks in an erasure-coded storage system running in a large datacenter environment is unique and more difficult, because it introduces three challenging dimensions to the problem–task scheduling over time, data source selection, and bandwidth allocation in each network segment to each background task. Existing scheduling algorithms designed for other problem domains consider mostly the first challenge and often assume homogeneous resources, e.g., processor scheduling across CPUs, MapReduce jobs across worker processes, etc. The recently proposed bandwidth reservation techniques, such as [19], can be used to better utilize network resource by allocating necessary amount of bandwidth to each task, but still did not consider source selection. Our problem is significantly different from these body of work. In an (n, k) erasure-coded system, scheduling background tasks requires selecting k out of n nodes as sources of a data flow. Furthermore, in a typical datacenter environment, at any given time the available network bandwidth for a tenant will significantly vary over time. Simply combining heuristics developed for the subproblems are insufficient for achieving an optimal performance, which calls for a joint optimization of all tasks over the "control knobs".

Well-studied heuristics such as Early Deadline First (EDF), First In First Out (FIFO) and Linear Programming (LP) do not take into account important constraints, such as network topology, data chunk placement and/or source chunk selection, thus resulting in unsatisfactory performance. In particular, FIFO is easy to apply in real systems, but has relatively low performance as observed in [20], [21]. EDF works well in networks with simple topologies [14], [15], [16]. But for datacenter networks that often employ a tiered-structure consisting of Top-of-Rack (TOR) and aggregation switches, EDF exhibits sub-optimal performance and fails to address data source selection when erasure coding is used (Sections 3 and 4).

In order to address these problems, we develop an online algorithm to maximize the number of tasks that successfully meet deadlines, under the constraints of data placement, network topology and available bandwidth. To optimally schedule each task, we need to jointly solve: (i) a chunk selection problem that determines the (erasure-coded) chunks used to generate background traffic, (ii) a bandwidth allocation problem that apportions bandwidth at TOR and aggregation switches among active tasks, and (iii) a scheduling problem that schedules tasks with respect to their deadlines. This optimization problem can be formulated as a mixed-integer optimization problem, which is proven to be NP hard (Section 3).

Our proposed algorithm leverages a novel metric called *Remaining Time Flexibility* (RTF) and jointly considers current network topology, source selection and bandwidth constraints. The RTF is the amount of time until a given task becomes infeasible with respect to its deadline. It measures the slackness of the starting time of a task and captures both task scheduling aspect (via task deadline and size) and source selection aspect (via available bandwidth). Intuitively, a task with higher RTF is less urgent (i.e., having a higher degree of flexibility with respect to both resource allocation and source selection) and can be postponed in the scheduling algorithm with relatively low risk of missing deadline. Our algorithm, called Linear Programming with Selected Tasks (LPST), is composed of three main steps. First, we choose k-out-of-n chunks from the most idle servers and racks. Second, we compute RTF for each background task with respect to its selected sources. Since the number of tasks ready to be scheduled can be large, we select a fewer number of relatively urgent tasks based on their RTF values. Third, we schedule the tasks through linear programming to determine the optimal bandwidth allocation for these tasks. The steps are then repeated for every task arrival and departure event in an online fashion to maximize network resource utilization (Section 4).

We evaluate existing algorithms as well as our proposed algorithm extensively in both simulation and real experiments in an OpenStack cluster. We demonstrate that the proposed algorithm can significantly improve the number of tasks finished before the deadlines under various combinations of arrival patterns, system parameters and resource availability. Our results show that, under certain scenarios, our proposed algorithm can perform 7x$\sim$10x better than the heuristics which blindly treat the infrastructure as a collection of homogeneous resources, and 21.7$\sim$65.9 percent better than the algorithms that only take into account the network topology. We also conducted a trace-driven simulation using a Google trace [22], [23]. The result was very promising—LPST perform 3x$\sim$17x better than others (Section 5).

2 Related Work

Our proposed algorithm - LPST (Section 3) is inspired by vast amount of related work. There exist very large body of work in process/packet scheduling algorithms [24]. We will not attempt to cover all existing related work but will discuss some directly related ones.

Our notion of remaining time flexibility is inspired by a classic scheduling algorithm called Least Slack Time First (LSTF) [25]. LSTF used a metric called *slack*, which is conceptually similar to RTF, to schedule tasks to a single or multiple processors and it can be easily applied to packet scheduling problem as well [26]. In S3 problem, similar to the reason that other simple heuristics will not work well, it is not enough to blindly apply LSTF since we need to additionally consider source selection and bandwidth allocation problems.

Aside from LSTF, many heuristic algorithms have been extensively studied in the community. The representative algorithms include Early Deadline First (EDF), First In First Out (FIFO), and Linear Programming (LP) and we discussed these algorithms with respect to S3 problem in Section 5.2. Some advanced algorithms based on these concepts are as follows. Algorithms based on FIFO has been applied for multicast traffic [20] and packet scheduling [21] to maximize system throughput. In [14], authors described a Global EDF algorithm to schedule parallel real-time tasks, which has provable performance bounds and overcomes task heterogeneity noted in [15], [16]. Lastly, using the model of a time-slotted system, traffic scheduling with deadlines can be formulated as a Linear Program (LP) problem. The complexity analysis of LP can be found in [27], [28], [29]. However, traffic

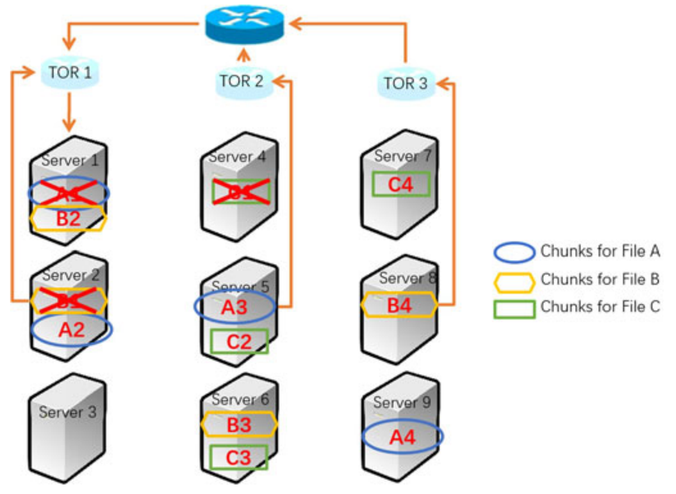


Fig. 1. An illustrative example with 3 racks and 9 servers. 3 files are stored using $(4, 2)$ code. None of the existing heuristics can complete all 3 repair tasks before their deadlines.

scheduling complexity grows quickly as network size and granularity increase [24], and it may lead to integer constraints when source selection and routing are involved [18].

Complementary to our work, substantial amount of work is proposed on reducing the amount of repair traffic in erasure coded storage systems. The list includes practical implementations that maintain local parities [30], [31] and novel codes that provide theoretical guarantees, e.g., MSR and MBR codes [32]. Since we assume MDS code in this paper and the majority of erasure codes used in practice maintain MDS property, our algorithm can be directly applicable to most work in this category.

3 System Model and Problem Formulation

We consider a datacenter storage system with one aggregator switch connecting u Top-of-Rack (TOR) switches. r storage servers ($\mathcal{R} = \{1, 2, \ldots, r\}$) are placed in u racks ($\mathcal{U} = \{1, 2, \ldots, u\}$), each of which is connected to a TOR switch. The traffic between servers in the same rack does not need to flow to the aggregator switch, while the traffic between servers in different racks needs to flow through two TOR switches and the aggregator switch. Each file i is stored using (n_i, k_i) erasure coding. We consider Maximum-Distance-Separable (MDS) codes, which ensures that any k_i out of n_i chunks are sufficient for reconstructing the file i.

3.1 An Illustrative Example

Consider the example in Fig. 1. We will illustrate that existing heuristics that work well for the sub-problems fail to achieve the optimal performance due to the lack of joint optimization over all “control knobs”. We consider a network with $u = 3$ racks and $r = 9$ servers. Three files A, B and C are stored using $(4, 2)$ erasure code. Each file is encoded into $n = 4$ chunks of different size $v_A = 6$ Gbits and $v_B = v_C = 8$ Gbits, allowing recovery from any $k = 2$ distinct chunks. At $t = 0$, one chunk of each file is lost and needs to be repaired before deadlines $d_A = 10$s, $d_B = 10.5$s and $d_C = 15$s, respectively. Suppose the link capacity is $CST = 2$ Gbps between servers and TORs, and $CTA = 3$ Gbps between TORs and the aggregator.

We consider 2 heuristic policies: (1) Use shortest path algorithm for source selection (i.e., select the chunk source that is closest to the destination where repair is to be done), and then use a first-fit heuristic to add tasks one-by-one (i.e., each receiving the least required bandwidth (Section 4) to meet its deadline) until no more tasks can be accommodated; and (2) Apply Earliest-Deadline-First to prioritize and schedule all tasks (i.e., selected tasks receive full remaining bandwidth after higher priority tasks are assigned) and then for this fixed schedule, select data sources in a way that it minimizes network congestion. We show that none of them are able to complete all 3 tasks before the deadlines.

Under Policy 1, source A_2 among others is selected to recover the lost chunk A_1, requesting a least required bandwidth of $v_A/d_A = 0.6$ Gbps on both servers 1 and 2. However, to recover a chunk B_1 on server 2, it needs to download 2 chunks of file B and requires an additional bandwidth of $2v_B/d_B = 1.52$ Gbps on server 2, exceeding bandwidth capacity of 2 Gbps. Thus, faulty chunks of files A and B cannot both be recovered before their deadlines. Under Policy 2, the three repair tasks are processed in the order of their deadlines. To balance network congestion, we choose servers 5, 9 (hosting chunks A_3, A_4) and servers 6, 8 (hosting chunks B_3, B_4) as sources to recover faulty chunks A_1 and B_1. Since task A_1 has the earliest deadline, it receives full bandwidth from $t = 0$s to $t = 6$s. Task B_1 can only utilize the remaining 1 Gbps bandwidth available at the aggregator switch until A_1 is recovered. It is easy to see that at $d_B = 10.5$s, there is still 1 Gb data remaining to be transfered for task B_1, resulting in a failure to meet its deadline.

However, completing all 3 tasks before the deadlines is indeed possible. Our key intuition is to consider RTF (Section 4), which measures the maximum available waiting time before a task becomes infeasible given its deadline. RTF captures both task scheduling (via task deadline and size) and source selection (via available bandwidth for each possible source) in the joint optimization. In particular, while B_1 has a later deadline, it has less slackness in scheduling, because its RTF (measuring maximum allowed waiting time before starting B_2) is $f_B = t_B - v_B/CST = 6.5$s, less than that of A_1, i.e., $f_A = 7$s. We use the same source selection as Policy 2, but give higher priority to task B_1 instead of A_1. It is easy to show that all 3 tasks are able to complete before the deadlines if we assign 2 Gbps to task B_1 from $t = 0$ to $t = 8$, allocate the maximum remaining bandwidth at aggregator (1 Gbps before B_1 completes and 2 Gbps afterwards) to A_1 and let tasks C_1 select minimally-congested servers 5,8. This strategy outperforms existing heuristics because RTF captures not only deadlines but also task sizes and bandwidth availability.

3.2 Problem Formulation

To maximize the number of background tasks that meet their desired deadlines, we consider a joint optimization in erasure-coded storage over 3 key dimensions: (i) selecting data sources, (ii) apportioning network resources among different background traffic flows with respect to network topology and deadlines, and (iii) scheduling multiple tasks to mitigate the “noisy neighbor problem”. These three sub-problems are closely coupled, resulting in an NP-hard problem (Section 3.3).

Our problem formulation is described as follows. Let $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$ denote a set of background tasks, such as backup, repair, and re-balance. Each task, A_i, is associated with a number of parameters, including n_i potential sources of data chunks (denoted as $o_{i,1} \in \mathcal{U}$, $o_{i,2} \in \mathcal{U}, \ldots,$ $o_{i,n_i} \in \mathcal{U}$), one destination (denoted as $p_i \in \mathcal{U}$), the number k_i of chunks to be retrieved, volume (denoted as v_i) for each chunk, task starting time (denoted as s_i), and task deadline (denoted as d_i). Task starting time and deadline are given in seconds, satisfying $0 \leq s_i \leq d_i$. To formally formulate this optimization problem, we consider a time slotted system. Suppose $y_{i,j}$ is a binary chunk selection variable, such that $y_{i,j} = 1$ if chunk j is selected to execute task A_i, and $y_{i,j} = 0$ otherwise. Since k_i data chunks must be selected, we have

$$\sum_j y_{i,j} = k_i, \ \forall i, \tag{1}$$

where the selection remains fixed while task i is running. For regenerating codes, it is possible that d < k chunks are required to repair a lost chunk, while it is also possible to access d > k chunks to minimize the repair bandwidth. In either case, it is equivalent to an erasure code with parameters (n,d) instead of (n,k). Since our proposed LPST algorithm works with arbitrary erasure codes, the joint scheduling and source selection problem remains the same, except for different parameters (n,d).

To count the number of successfully completed tasks, we use a binary variable z_i, which is 1 if task A_i is finished before the deadline, and 0 otherwise. Let $x_{t,i,j}$ be the bandwidth assigned in time slot t to the data flow transferring chunk j of task A_i. If the task is successfully completed before a deadline d_i, all of the k flows should finish before d_i, implying a deadline constraint for successful tasks:

$$\sum_{t=s_i}^{d_i} x_{t,i,j} y_{i,j} \geq v_i, \ \text{if } z_i = 1, \forall i, \forall j. \tag{2}$$

Since each source-destination pair has a predetermined route, for a given set of tasks, we use RC_g to denote the set of tasks/chunk flows traversing a (TOR or aggregator) switch g, i.e., $(i,j) \in RC_g$ if flow of chunk j of task i uses switch g. Similarly, SC_h is the set of tasks/chunk flows using a server h. Further, each TOR has capacity limit CTA, and each server has capacity limit CST. The link capacities, CST and CTA, are actually the amount of bandwidth available for background traffic optimization, i.e., the maximum link capacity minus the bandwidth assigned to foreground traffic. Thus, we have the following capacity constraints:

$$\sum_{(i,j) \in RC_g} x_{t,i,j} y_{i,j} \leq CTA, \ \forall g, t \tag{3}$$

$$\sum_{(i,j) \in SC_h} x_{t,i,j} y_{i,j} \leq CST, \ \forall h, t. \tag{4}$$

Our goal is to maximize the number of tasks that can be successfully completed before deadline in erasure-coded storage. This is formulated as a joint Scheduling and Source Selection (denoted as $S3$) problem, i.e.,

$$\max \quad \sum_i z_i \tag{5}$$

$$\text{s.t.} \quad \sum_j y_{i,j} = k_i, \ \forall i \tag{6}$$

$$\sum_{t=s_i}^{d_i} x_{t,i,j} y_{i,j} \geq v_i z_i, \ \forall i, \tag{7}$$

$$\sum_{(i,j) \in RC_g} x_{t,i,j} y_{i,j} \leq CTA, \ \forall g, t \tag{8}$$

$$\sum_{(i,j) \in SC_h} x_{t,i,j} y_{i,j} \leq CST, \ \forall h, t \tag{9}$$

$$var. \quad x_{t,i,j} \geq 0, \ y_{i,j} \in \{0,1\}, \ z_i \in \{0,1\}. \tag{10}$$

Here the deadline constraint (7) is exactly (2) for successful tasks with $z_i = 1$, and is superfluous when $z_i = 0$. Note that replication can be considered as a special case of our proposed optimization with $k_i = 1$, i.e., the entire file is replicated across the network.

3.3 Proof of NP-Hardness

Theorem 1. *The proposed $S3$ Problem is NP-hard.*

Proof. We show that if the $S3$ Problem can be solved in polynomial time, then the maximum independent set problem can also be solved in polynomial time, which contradicts the known NP-hardness of maximum independent set problem. For some small $\epsilon > 0$, we consider a special case of the $S3$ Problem with following simplifications:

1. There is only one rack;
2. All chunks have equal size v;
3. Equal link capacity CST from each server to TOR.
4. All tasks have equal deadline $d = v/CST + \epsilon$.

Our formulation implies that only 1 chunk can be transferred from any server before deadline. It remains to prove that if the $S3$ Problem can be solved in polynomial time, so is the maximum independent set problem.

We consider a given instance of maximum independent set problem and converts it to an $S3$ Problem in the special case. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an independent set is a set of vertices in the graph, such that no two vertices are connected by an edge. A maximal independent set problem is to find an independent set that has the largest number of vertices. It is well known as an NP-hard problem [33], [34]. Given a maximum independent set problem, we convert it into a $S3$ Problem with $m = |\mathcal{V}|$ files and $r = |\mathcal{E}|$ servers as follows: We use a file to represent each vertex in $\mathcal{V}$. If there is an edge connecting $i, j \in \mathcal{V}$, we let files i, j share a common server by placing a single chunk of each file into the server. Thus, each file i has n_i chunks, which is equal to the degree of vertex $i\mathcal{V}$ in $\mathcal{G}$. Finally, we choose erasure codes $n_i = k_i$ for any file i. It means that all n_i chunks must be retrieved to reconstruct the file before deadline.

Assume that all files need to be backed up on a server with large enough bandwidth. Our goal in $S3$ is to maximize the number of such $m = |\mathcal{V}|$ tasks that can be completed before deadline. It is easy to see that if we can solve this $S3$ problem, we can also solve the maximum independent set problem. Clearly, no 2 tasks can be completed at the same time if they have 2 chunks sharing a common server (due to the deadline and link capacity constraint, only allowing 1 chunk to be retrieved). Maximizing the number of successfully-completed tasks is equivalent to finding the maximum independent set on $\mathcal{G}$. Since the maximum independent set problem is NP-hard, we conclude that our problem is also NP-hard. □

4 LPST Algorithm Design

In this section, we describe our proposed algorithm, called Linear Programming for Selected Tasks (LPST), which harnesses resource-aware chunk selection, deadline-aware task prioritization, and bandwidth optimization via linear programming. As a result, LPST maximizes the number of tasks that can meet their deadlines. We compare LPST to a set of heuristic algorithms[2] and their qualitative descriptions are presented in Section 5.2. Quantitative comparison results are in Section 5. We devide the S3 problem into 3 subproblems—selecting sources, scheduling tasks and assigning bandwidth.

Selecting Sources (Phase I). When a new task arrives to the system, the algorithm needs to decide which sources will be used for the task. The source selection process will affect the resource availability for other tasks in the system. Recall that, in (n_i, k_i) erasure coding used for task i, we need only k_i chunks out of n_i, which will be enough to reconstruct the original data. If one chunk is lost and k_i chunks are chosen by task i, a future task might be able to utilize other $n_i - k_i - 1$ chunks without worrying about the execution status of the previous task. LPST implements a source selection policy that finds the first k_i subtasks that make the network least congested. in order to better utilize network resources. Many well-known distributed storage systems want to distribute data uniformly across the available machines. This ensures the scalability of the system and provides a certain level of reliability guarantees. For instance, Ceph [1] uses CRUSH algorithm [35] to make each OSD equally contribute to the client load. Swift [2], HDFS [36], and Ambry [37] have similar design rationals, i.e., either placing data as equally as possible or regularly rebalancing data, to achieve the same high level goals.

Suppose a task A_i arrives to the system. Then A_i is split into k_i subtasks, each of which has a distinct source. Each subtask $A^{'}_{i,s}$ (s=1..k_i) has 5 properties: a) source ($o^{'}_{i,s}$), b) destination ($p^{'}_{i,j} = p_i$), c) volume ($v^{'}_i = v_i$), d) starting time ($s^{'}_i = s_i$), and e) deadline ($d^{'}_i = d_i$). Note that while each subtask $A^{'}_{i,s}$ has its own selected source, all subtasks belonging to A_i must be completed before d_i to meet a common deadline. For each of the k_i subtasks, we calculate its *least required bandwidth (LRB)*, defined by the minimum amount of bandwidth that is necessary to finish the task before the deadline. Let t be the current system time. LRB can be calculated using the following equation.

$$LRB_i = v_i/(d_i - t). \tag{11}$$

Then, for the corresponding servers or TORs in the path, we add LRB_i to their congestion factors. Then we calculate the congestion factors for all subtasks, and we select k_i sources with least fulfilled links (smallest congestion factor).

Prioritizing Tasks (Phase II). Once the sources are chosen, we could generate a plan on how we may allocate bandwidth for the tasks to satisfy our objective, e.g., maximizing network utilization of our datacenter. However, blindly applying existing optimizing technique, such as linear programming, is likely to cause a scalability problem (Section 5). Therefore, in LPST we first sort all subtasks based on a metric, called *remaining time flexibility (RTF)*, which quantifies the flexibility in scheduling a task with respect to its deadline and resource availability, reflecting how emergent the task is. After a list of admitted tasks are identified, a linear programming problem is solved to optimize bandwidth allocation for maximizing network utilization for the admitted tasks.

Algorithm 1. LPST Algorithm

```
1  // Phase I: Source Selection Procedure
2  foreach task i do
3      Least required bandwidth: $LRB_i = v_i/(d_i - t)$;
4      Sort $w_i$ candidate sources by the largest congestion
         factor in each path from source to destination;
5      Find $k_i$ source servers with least fulfilled path;
6      Create $k_i$ new subtask $A^{'}_{i,s}$;
7      Add $LRB_i$ to congestion factor of links in each subtask's
       path;
8  end
9  // Phase II: Selecting Emergent Tasks
10 foreach subtask i do
11     Calculate RTF $f_i = \min_s\left(d^{'}_i - max(t, s_i) - v^{'}_n/C_{o_{i,s},p_i}\right)$;
12 end
13 Initialize $\mathcal{T} = \{\}$, remaining bandwidth for each link;
14 Find task $i$ with smallest $f_i$;
15 while task i is feasible w.r.t. remaining bandwidth do
16     $\mathcal{T} \leftarrow \mathcal{T} \bigcup \{A_i\}$
17     Assign initial bandwidth $b_i = LRB_i$;
18     Update remaining bandwidth;
19     Find next task $i$ with smallest $f_i$;
20 end
21 // Phase III: Optimize bandwidth for admitted tasks in $\mathcal{T}$ ;
22 Solve the following optimization problem using LP;
23 max $\sum_{i:A_i\in\mathcal{T}} b_i$
24 s.t. $\sum_{(i,s)\in RC_g} b_i \le CTA, \forall g$
25     $\sum_{(i,s)\in CS_h} b_i \le CST, \forall h$
26     $b_i(d_i - s_i) \ge v_i \; \forall i$
27 var. $\{b_i, \; \forall i \in \mathcal{T}\}$
```

In particular, for subtask $A^{'}_{i,s}$, a chunk of size $v^{'}_i$ needs to be transferred, from source server $o^{'}_{i,s}$ to destination server $p^{'}_i$, which has pre-determined route with maximum available capacity $C_{o_{i,s},p_i}$. The task starting time is s_i and deadline is $d^{'}_i$. Then RTF $f^{'}_{i,s}$ of the subtask $A^{'}_i$ can be calculated as follows.

$$f_{i,s}{}' = d^{'}_i - max(t, s_i) - v^{'}_n/C_{o_{i,s},p_i}, \tag{12}$$

where t is current timestamp and $C_{o_{i,s},p_i}$ is the maximum available link capacity from source server $o^{'}_{i,s}$ to destination server $p^{'}_i$. Next, the RTF of task A_i is defined as the minimum RTF of all its subtasks, i.e.,

$$f_i = \min_s f^{'}_{i,s}. \tag{13}$$

Intuitively RTF f_i measures the maximum allowed delay to begin processing task A_i, in order to meet its deadline. If f_i value is smaller, the task is more emergent and we may need to schedule it right away by delaying some other tasks that have higher RTF values.

2. Some of these heuristics are extensively studied in the community. Note that the proof of NP-hardness is provided in Section 3.3.

TABLE 1
Table of Key Notations

m	number of tasks
(n, k)	erasure code parameters
$\mathcal{A}$	A set of m tasks $A_1, \ldots, A_m$
RC_g	A set of tasks traversing TOR/aggregator switch g
SC_h	A set of tasks using server h
r	number of storage servers
u	number of racks
CST	Link capacity from servers to each TOR
CTA	Link capacity from each TOR to the aggregator
$x_{t,i,j}$	Bandwidth assigned at t to send chunk j of task A_i
w_i	Number of candidate sources/chunks for task A_i
z_i	Whether task i is completed before deadline
$o_{i,1}, \ldots, o_{i,w_i}$	Candidate sources/chunks for task A_i
$o^{'}_{i,s}$	Selected source/chunk for sub-task $A^{'}_{i,s}$
p_i $(p^{'}_i)$	Destination of task A_i (sub-task $A^{'}_{i,s}$)
v_i	Volume (chunk size) of task A_i
d_i	Deadline of task A_i
s_i	Starting time of task A_i
LRB	Least required bandwidth of task A_i
f_i $(f^{'}_{i,s})$	RTF of task A_i (sub-task $A^{'}_{i,s}$)

TABLE 2
Illustration of how our LPST Algorithm Works for the Example in Fig. 1, Section 3

Time	Remaining Time Flexibility	Task status
0	$A_3(7)$,$A_4(7)$, $B_3(6.5)$ $B_4(6.5)$,$C_2(11)$,$C_3(11)$	$A_3(6,0.6)$,$A_4(6,1.4)$,$B_3(8,0.76)$ $B_4(8,1.24)$,$C_2(8,1)$,$C_3(8,1)$
6.28	$A_3(2.6)$,$B_3(2.6)$,$B_4(4.17)$ $C_2(7.86)$,$C_3(7.86)$	$A_3(2.23,1.5)$,$A_4(0,C)$,$B_3(3.23,1.5)$ $B_4(0.21,0.05)$,$C_2(1.72,0.5)$,$C_3(1.72,0.5)$
7.77	$B_3(2.23)$,$B_4(1.73)$ $C_2(5.27)$,$C_3(5.27)$	$A_3(0,C)$,$B_3(1,0.5)$ $B_4(2,1.5)$,$C_2(0.98,1)$,$C_3(0.98,1)$
9.1	$B_3(1.235)$	$B_3(0.33,2)$,$B_4(0,C)$,$C_2(0,C)$,$C_3(0,C)$
9.76		$B_3(0,C)$, all tasks complete

Finally, we rank all tasks according to their RTF in ascending order, and admit tasks one-by-one until no more task with higher RTF can be added.

Assigning Bandwidth (Phase III). After we get a final list of feasible tasks, we formulate a network optimization to assign bandwidth b_i for each task by maximizing network link utilization. This is shown in Phase III of Algorithm 1. While this step does not directly affect the number of tasks that are completed before deadline, it maximizes resource utilization and thus reduces the overall completion time required by currently admitted tasks. This has two benefits. First, when we use the proposed LPST algorithm in an iterative fashion, optimizing bandwidth utilization allows us to accommodate more tasks by re-running the procedure in Phase I and II. Second, this is particularly important in an online setting—by completing the current, admitted tasks as fast as possible, we can make more resources available for new tasks that arrive in the future. The bandwidth assignment in Phase III is solved as a linear programming problem with network capacity and deadline constraints. The admitted tasks are guaranteed to meet their individual deadlines.

Supporting Different Network Topologies. Although in this paper we formulate our optimization for a hierarchical datacenter network topology involving TOR and aggregator switches, the results can be readily extended to arbitrary topologies such as fat-tree or Bcube [38], [39]. In particular, source selection (Phase I) and bandwidth assignment (Phase III) need to reflect updated link capacity constraints due to new network topologies, while task prioritization (Phase II) remains the same. More complicated network topologies, such as B-cube or fat-tree, may introduce more link capacity constraints, but they are still linear constraints and can be solved by linear programming. Since LPST uses task prioritization, the complexity of linear programming will still be limited due to small number of variables.

Complexity Analysis of LPST Algorithm. In this section, we analyze computational complexity of LPST algorithm.

Remark. The time complexity for Phase I is $O(m)$.

As shown in Table 1, m is the number of tasks and k is the number of chunks to be transmitted. For m tasks, we need to make source selection one by one, so there are m iterations in outer loop. As for each task A_i, if one chunk is lost, we need to select k_i sources from w_i remaining source options and update the link congestion status for other tasks, which has $O(n^2)$ operations. Notice that the parameter n is a very small number (no more than 25) in a typical erasure coded storage systems. Therefore, $O(n^2)$ can be replaced by O(1). Thus, the time complexity for source selection is $O(m)$.

Remark. The Phase II has time complexity O(m′logm′).

There are $m^{'}$ selected subtasks to be transmitted. Ranking mk transmissions by their remaining time flexibility has time complexity O(m′logm′).

Remark. The linear programing block has time complexity $O((u + r + m')log(m')/\epsilon^2 + m')$.

According to [40], given a linear programming problem with a constraint matrix that has n non-zeros, r rows, and c columns, a proposed algorithm (with high probability) computes feasible primal and dual solutions whose costs are within a factor of 1 + ϵ of opt (the optimal cost) in time $O((r + c)$ $log(n)/\epsilon^2 + n)$. In our linear programming block, the variables are the assigned bandwidth for transmission tasks selected from previous phase. In the worst case, all of the mk tasks are selected for scheduling. So the number of variables is at most mk. Each server to TOR link and TOR to the aggregator link has one constraint. So there are $u + r$ rows in the constraint matrix. At most, there are mk columns in the matrix if all tasks are placed in the same server. So in the worst case, the linear programing block has time complexity $O((u + r + m')$ $log(m')/\epsilon^2 + m')$. Since selected tasks are only part of original tasks, this procedure will not be slow and can adapt to more complicated network topology.

In summary, the time complexity of the entire algorithm is $O(m + m'logm' + (u + r + m')log(m')/\epsilon^2 + m')]$ The illustration of the algorithm is presented in Algorithm 1.

Example. We use the same example in Fig. 1, Section 3 to demonstrate how the proposed LPST algorithm jointly solves the S3 optimization over task scheduling and source selection. The results are shown in Table 2. We adopt the following notation: For RTF, we use $A_i(f)$ to represent that the task with source A_i currently has remaining time flexibility f. Similarly, we use $A_i(v, b)$ to represent a task with remaining volume v and status b, which could be the bandwidth value if it is active or a code (i.e., W,C,F) denoting waiting, completed, and failed task status.

For source selection, we select A_3 and A_4 as sources for file A at $t = 0$. The least required bandwidth for transmitting file A is 6 Gb/10s = 0.6 Gbps. We add 0.6 Gbps to the congestion factor of the links in the path of transmitting A_3 and A_4. For file B, the largest congestion factor within B_2, B_3 and B_4's path is 1.2 Gbps, 0.6 Gbps and 0. So we select B_3 and B_4. Then we update congestion factors. For file C, the largest congestion factor for C_2, C_3 and C_4 is 0.6 Gbps, 0.76 Gbps, and 0.76 Gbps. Select C_2 and C_3.

At $t = 0$, B_3 and B_4 have the smallest RTF. We assign LRB (least required bandwidth) of 8/10.5=0.76 Gbps to them initially and update remaining bandwidth for links in the path of B_3 and B_4. Then A_3 and A_4 have secondary RTF and LRB is 0.6 Gbps. Then we assign LRB and update the remaining bandwidth for links. Then C_2 and C_3 are assigned LRB of 0.53 Gbps. To make full use of available bandwidth resources, we re-optimize bandwidth for the selected tasks as described in the last phase of the propsoed LPST algorithm. The resulting bandwidth is shown in Table 2. At $t = 6.28$, A_4 completes. Then we repeat the calculation. At time instant 9.76 second, all tasks complete successfully.

5 Evaluation

5.1 Methodology, Simulator, and Experimental Setup

As discussed in Section 3, our problem space has many dimensions to explore. To properly evaluate the performance of LPST and other algorithms, we conducted extensive *simulations* and validated the results with *experiments in a real cloud environment*.

To this end, we built a custom simulator and a prototype implementation of all algorithms in an Openstack-based cluster [41]. In addition, we implemented a *task generator* to feed tasks into both the simulator and the prototype system for experiments. The simulator is event-driven and written in Java. It takes the *task generator*'s output as input and simulates the behavior of various algorithms. It captures essential resource constraints including network topology, bandwidth limitation, task deadlines, and erasure-code source selection. We also implemented a prototype system to validate simulation results. Similar to our simulator, our prototype system takes the *task generator*'s output as input. However, it actually schedules the tasks to a real cloud environment managed by Openstack, and consequently generates real packet transmissions among VMs. Our prototype consists of two subcomponents — *task scheduler* as a control plane and a rsync [42] based data plane.

The cloud environment that we conducted experiments has 16 physical servers, with each server having a 10 Gbps network connectivity to a single TOR switch. We created 30 VMs to construct a virtual topology with 3 racks and one aggregation switch. The constructed topology is similar to Fig. 1, with each rack consisting of 10 servers. Each VM has 2 VCPUs, 4 GB RAM and 40 GB virtual disk drive.[3] We use rsync to limit the bandwidth usage of each scheduled task in our experiments. Whenever an event occurs according to a given algorithm, e.g., a task arrives or completes, we pause ongoing background operations is these VMs, perform computations based on the scheduling algorithm, and send remote ssh commands to VMs to resume data transmissions for background jobs. Scheduling parameters such as allocated bandwidth and transmission time are piggybacked on these commands and applied to rsync arguments. Rsync uses delta encoding and supports "suspend" and "resume" operations for these tasks. When a paused task is resumed, rsync checks the difference and transmits the remaining part of the data.

3. Due to quota issues, we were limited to 40GB drives. But the results are not fundamentally affected by small drive size.

All results presented in this paper in each point are average values computed over 1000 tasks. We evaluate different algorithms using three metrics—number of tasks completed by the deadline, remaining volume, and link utilization. *Remaining volume* refers to the amount of data in GB, whose transmission to the destination server was not completed by the deadline. *Link utilization* is the ratio of the total amount of data that can be transferred through a given network link to the total amount of data that was actually transferred through that link. In all settings, erasure-coded chunks are placed uniformly following the best practices of many distributed storage system in a real world as discussed in Section 4.

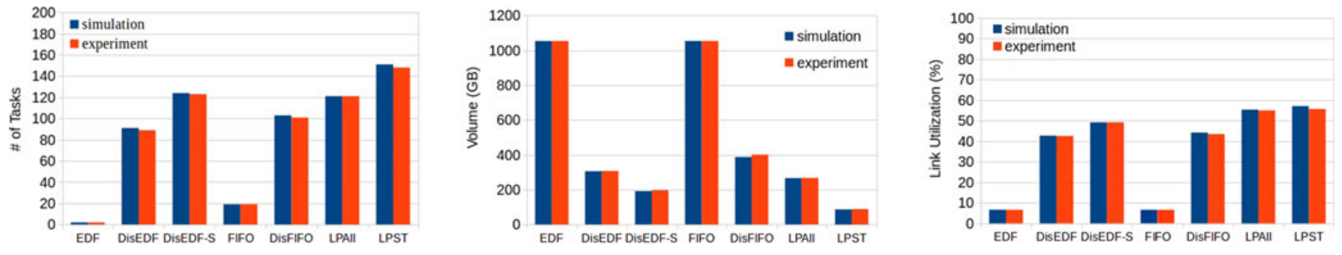
5.2 Competing Algorithms

We compare LPST against several variants of well-known heuristic scheduling algorithms. The competing algorithms that we considered in this paper are three-fold—FIFO and its variants, EDF and its variants, and Linear Programming.

FIFO Family. Due to its simplicity, First In First Out (FIFO) scheduling algorithm is widely used in different problem domains. The algorithm schedules a task to the first available resource in a sequential manner. FIFO has an obvious inefficiency when two consecutive tasks share the same network link. In Fig. 1, consider the case in which the task A_1 is transmitting data from server 2 to server 1 and the task A_2 is sending data from server 8 to server 5. In FIFO, A_2 will need to wait until A_1 completes. To address this issue, we come up with a disjoint version of FIFO (DisFIFO). In DisFIFO, the tasks that do not share network links can be scheduled at the same time, and consequently result in better performance. Lastly, all algorithms in FIFO family choose sources randomly in erasure-coding case.

EDF Family. Earliest Deadline First (EDF) algorithm is also well-studied in the scheduling literature. In our problem setting, the EDF algorithm has the same problem like FIFO. So we developed DisEDF using the similar technique. The difference between EDF and DisEDF is exactly the same as that between FIFO and DisFIFO.

Linear Programming. We utilize a recent advance in datacenter networking, i.e., bandwidth reservation, to devise an algorithm called Linear Programming applied on All tasks (LPAll). Whenever a new task arrives to the system or a task finishes, LPAll assigns bandwidth to a given set of tasks using the linear programming technique. The formulation is same as that of LPST bandwidth allocation scheme, i.e., the objective function is to maximize bandwidth utilization under link capacity and task deadlines constraints.



(a) # of Completed Tasks (b) Remaining Volume (c) Link Utilization (Server↔TOR)

Fig. 2. Experimental results in a real Openstack cloud environment match very well with the simulation results. LPST outperforms all competing algorithms.

TABLE 3
Parameters used for Simulation & Experiments

	# of Tasks	Erasure Code	Arrival Rate(s^{-1})	Chunk Size(MB)	Link (Mbps)	Deadline
Baseline	1000	(9,6)	Poisson, 0.1	64	500/1500	+LRT * 10
Each Phase Contribution	1000	(9,6)	Poisson, 0.1	64	500/1500	LRT * 10
Foreground Task	1000	(9,6)	Poisson, 0.1	64	0∼ 60%*500/1500, Uniform Distribution	LRT * 10
(9,6), (14,10) Erasure Code	1000	(9,6)	Poisson, 0.1	64	500/1500	LRT * 10
Chunk Size	1000	(9,6)	Poisson, 0.1	64∼2048	500/1500	LRT * 10
Arrival Rate	1000	(9,6)	Poisson, 1/30∼2	64	500/1500	LRT * 10
Deadline	1000	(9,6)	Poisson, 0.1	64	500/1500	LRT * (2, 5, . . ., 10)

+*Least Required Time.*

5.3 LPST Performance and Validation of Simulation Results

Fig. 2 shows results from both simulation and real experiments. Table 3 shows parameters used for simulation and real experiments. The "baseline" row shows the common parameters while other rows show how the variable parameters are changed. We used (9,6) erasure code[4], which is a popular erasure-code scheme used by many practical systems [7], [43].

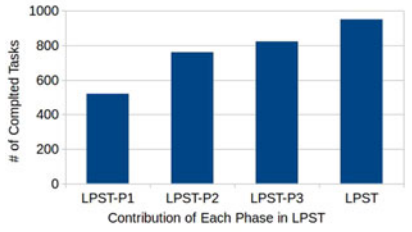
As shown in Fig. 2a, LPST completes significantly greater number of tasks within deadline than other algorithms. For example, compared to FIFO and EDF, LPST completes 7x and 70x more tasks within deadline. Compared to disjoint versions of these algorithms (Section 4), LPST still shows 46.6 to 65.9 percent better performance. Compared to even more optimized algorithms, such as DisEDF and LPAll, LPST completes 21.8 and 24.8 percent more tasks, respectively. Fig. 2b shows that the amount of data not transmitted by background jobs within deadline is significantly lower in LPST than in other algorithms. Fig. 2c shows network utilization by all algorithms averaged over all network links. Since LPST uses network resources efficiently, it is able to parallelize background tasks in disjoint network links, resulting in better performance.

It is not surprising that naive EDF and FIFO algorithms—who do not consider network topology, source selection and/or bandwidth constraints—exhibit poor performance. Notice that they utilize network link capacities poorly (Fig. 2c, and thus fail in parallelizing data flows across disjoint network links. Although FIFO amd EDF have similar amount of *remaining volume*, FIFO completes more tasks within deadline. This is because a scheduled task in EDF can be interrupted by tasks that arrive later, but have shorter deadline. This can impact the number of tasks that finish within deadline negatively. These results show that it is important to consider network topology when designing a scheduling algorithm. All enhanced algorithms that take into account network topology, e.g., DisFIFO and DisEDF, perform much better than corresponding original algorithms. For example, DisEDF and DisFIFO finish 45x and 5x more tasks than EDF and FIFO, respectively. Another important factor that affects the performance of a scheduling algorithm is the proper selection of erasure-code sources. For example, compared to DisEDF increases the number of tasks completed within deadline by 36 percent and reduces *remaining volume* by 58 percent, while increasing bandwidth utilization slightly. Notice that LPST is still better than DisEDF. The reason is that, although DisEDF considers network topology and source selection, it does not control bandwidth allocations among tasks scheduled in the same time slot. In contrast, LPST assigns appropriate amount of bandwidth to each task, resulting in better utilization of network resources for all tasks. Improving the utilization of cloud infrastructure resources is important for cloud service providers.

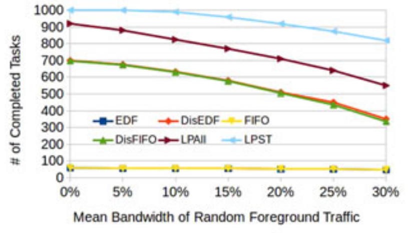
It is interesting to observe that LPST performs better than LPAll, which focuses on optimizing bandwidth utilization. Note that both LPAll and LPST have very similar bandwidth utilization. However, LPAll does not consider task deadlines, which results in a significant degradation in performance.

It should be noted that our goal in presenting performance of enhanced algorithms like DisEDF and DisFIFO is to show that we need to consider all three factors—smart source selection, appropriate network bandwidth allocations, and deadline-aware scheduling—*together* in order to schedule background tasks efficiently. These are the novel

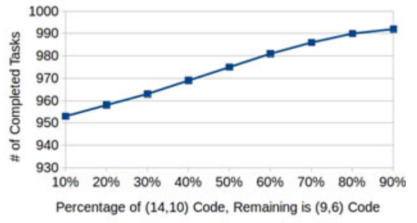
4. Note that we use $(9,6)$ and $(6+3)$ formats interchangeably.



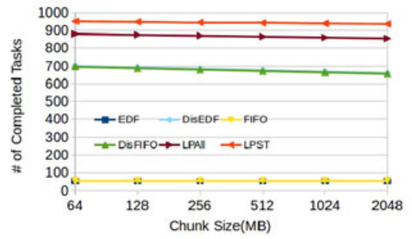
(a) Comparison of each phase's contribution of LPST



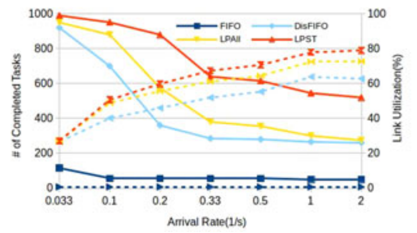
(b) Increasing foreground task influence



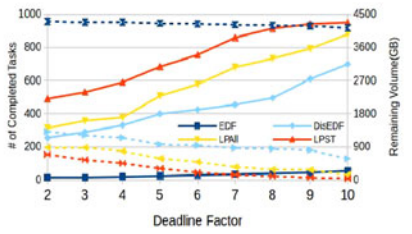
(c) Combining (9,6) and (14,10) Erasure Codes



(d) Increasing chunk size has little influence on performance



(e) Increasing task arrival rate reduces completed task number but increases link utilization. For the same algorithm, the solid line that indicates the number of completed tasks has the same color as the dashed line that indicates the link utilization.



(f) Increasing deadlines increases completed task number and decreases remaining volume of failed tasks. For the same algorithm, the solid line that indicates the number of completed tasks has the same color as the dashed line that indicates the remaining volume of failed tasks.

Fig. 3. Sensivity analysis vis simulations.

aspects of scheduling background traffic in practical datacenter environments.

Fig. 2 also validates that our simulation results closely resemble real experiment results. The difference between simulation and experimental results are negligible, less than 2.2 percent. We also performed 10000 other tasks by using different parameter settings(not included in this paper for brevity) and the difference between those results and corresponding simulation results was in a similar range. It should be noted that we cannot cover all the parameter spaces using experiment because, in real experiments, handling only 200 tasks takes about 3$\sim$4 hours, mainly due to wide spanning task deadline settings.

5.4 Sensitivity Analysis via Simulations

Next we thoroughly investigate the effect of several parameters on the performance of various agorithms. We use simulations for these evaluations. Please note that simulation results match real experiment results quite well. To make simulation results more realistic, we use 64MB(which is used in Google clusters) as default chunk size, and use (9,6) erasure code(which is used in Google ColossusFS) and (14,10) erasure code (which is used in Facebook HDFS) as erasure code patterns. Then, we use parameters from Google trace for validation simulation. Overall, for almost all parameter space we explored, we found that LPST either outperforms competing algorithms, or performs at least as good as other algorithms. The parameters used for the simulation runs are described in Table 3.

Comparison of Each Phase's Contribution of LPST. We divide the problem into 3 subproblems - selecting sources, scheduling tasks and assigning bandwidth. Fig. 3a shows the contribution of solving each subproblem in the joint optimization. We consider 3 new algorithms, each of which is constructed by keeping our proposed algorithm for 1 subproblem and applying some simple heuristics to the other 2 subproblems. The 3 heuristics we use to replace our algorithms for different subproblems are: 1. selecting sources: randomly pick sources; 2. scheduling tasks: tasks with earlier start time are executed early; 3. assigning bandwidth: assign the least required bandwidth to each task. As shown in Fig. 3a, LPST-P1 finishes 38.61 percent less tasks, while LPST-P2 and LPST-P3 respectively reduce 17.35 and 12.89 percent completed tasks. Thus, solving the source selection problem has the least contribution, and assigning bandwidth has the largest contribution followed by scheduling.

Foreground Task Influence. We conduct a simulation to show the influence introduced by random and time-varying foreground tasks in Fig. 3b. Each link capacity has a maximum capacity of 500 Mbps as CST and 1500 Mbps as CTA. Random and time-varying Foreground tasks are occupying link capacity. For those foreground tasks in each link, their bandwidth utilization are randomly generated between $0\sim10\%, 0\sim20\%, \ldots, 0\sim60\%$ from a uniform distribution from time to time. Thus, the mean bandwidth of random foreground tasks in each link is from $5\%\sim30\%$. Whenever there are large foreground traffic change, or new coming background task, or completed background task, we do optimization calculation. LPST always completes the largest amount of tasks than other competing algorithms. Although the performance of algorithms decreases with more foreground traffic, the relative benefit of LPST becomes higher, demonstrating the importance of joint S3 optimization in heavy and complicated network traffic environment. For example, when foreground tasks occupy 5 percent link capacity on average, LPST only completes 13.63 percent more tasks than LPAll, but this difference enlarges to 48.91 percent when foreground tasks occupy 30 percent link capacity on average.

Combination of Two Realistic Erasure Codes. We show with the performance of LPST with combination of 2 realistic erasure codes - (9,6) and (14,10) in Fig. 3c. (9,6) erasure code is applied in Google ColossusFS and (14,10) erasure code is applied in Facebook HDFS[44]. We change the percentage of (9,6) and (14,10) erasure code among tasks, from 90 percent tasks using (9,6) erasure code and 10 percent tasks using (14,10) erasure code, to 10 percent tasks using (9,6) erasure code and 90 percent tasks using (14,10) erasure code. The performance improves as more tasks employ (14,10) erasure code. This is because (14,10) erasure code provides more flexibility in source selection, but the benefit is not significant due to relatively low contribution of source selection phase in LPST shown in Fig. 3a.

Chunk Size. We examine the sensitivity of various algorithms with respect to erasure-coded chunk size. Fig. 3a shows that LPST performs consistently better than other algorithms in the entire range of data size we used in our experiments. LPST on average completes 8.76 percent more tasks than LPAll, 38.75 percent more tasks than DisEDF and DisFIFO, and 1615.46 percent more tasks than EDF and FIFO. DisEDF has similar performance as DisFIFO, and so do EDF and FIFO. In other words, unlike other resources tightly coupled with the infrastructure, data size has less impact on the relative performance of these algorithms. This is mainly because data size impacts all algorithms in a similar way. Data size does not impact the factors for which these algorithms are designed. Specifically, data size does not directly affect parameters like network topology, source selection, bandwidth allocation, and deadlines.

Arrival Rate. The rate at which background jobs arrive in the system is an important parameter that can affect performance. We conduct a simulation study with different arrival rates while fixing other parameters. The number of completed tasks and link utilization(dashed lines) are shown in Fig. 3e. For the same algorithm, solid line that indicates the number of completed tasks has the same color as dashed line that indicates the link utilization. To make the figure concise, the performance of DisEDF and EDF are not shown, since they have very similar performance as DisFIFO and FIFO. The impact of arrival rate is quite significant, e.g., the number of completed tasks can be degraded by 48.36 percent under demanding arrival rates, but the link utilization can be upgraded by 58.02 percent.

Not surprisingly, as the arrival pattern becomes more sparse, the performance gap between LPST and greedy alternatives gets narrower. In the most sparse arrival pattern we tried (arrival rate of 0.033 tasks per second), many algorithms perform equally well. However, many algorithms complete less tasks but have better link utilization when arrival rate pressure gets higher. In the most dense arrival pattern, LPST completes 89.09, 99.12 and 1041.63 percent more tasks than LPAll, DisFIFO and FIFO. By comparing the completed task number and link utilization of LPST and LPAll, we can see LPST completes much more tasks than LPAll, although the link utilization is close. This is because LPAll optimizes the bandwidth allocation to have high link utilization without considering scheduling, and thus will transmit a lot of non-urgent data.

Deadline. Next we examine the impact of task deadlines. We set our deadline as *(deadline factor)* $\times$ *(least required time (LRT))*. LRT is a fixed value and can be calculated using $\frac{DataSize}{FullLinkCapacity}$. For a given LRT, a higher deadline factor means there is more time for scheduling. A smaller deadline factor

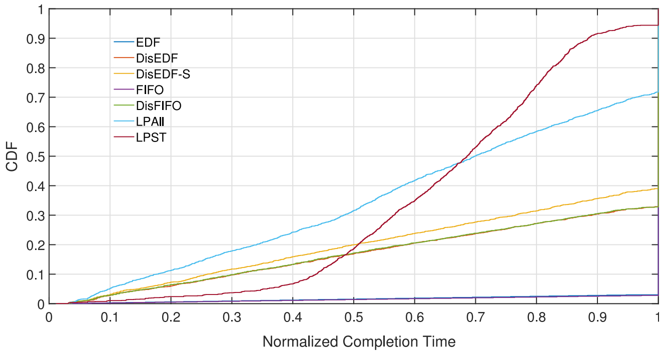


Fig. 4. CDF graph for normalized completion time.

means there is greater urgency in scheduling tasks to meet their deadlines.

Fig. 3f shows the number of completed tasks and remaining volume of failed tasks(dashed lines). For the same algorithm, solid line that indicates the number of completed tasks has the same color as dashed line that indicates the remaining volume of failed tasks. To make the figure concise, DisFIFO and FIFO are not shown since they have similar performance of DisEDF and EDF. Overall LPST still performs significantly better than other algorithms and the advantage gets larger for tighter deadlines.

By comparing the completed task number and remaining volume of failed tasks of LPST and LPAll, we can see LPST complete much more tasks than LPAll although the remaining volumes of both algorithms are similar. This is because LPAll optimizes the bandwidth allocation without considering scheduling, and thus will transmit a lot of data but miss deadlines and leave small amount of remaining volume of many tasks that have tight deadlines or crowded links.

5.5 Google Trace

We further conduct additional evaluation to validate the ability of our algorithm to schedule file access requests with real-world Google trace arrival patterns. Google trace contains all types of workloads(including but not limited on "background traffic" like repair, backup and rebalancing traffic) running on Google compute cells. Since it does not indicate individual task type, we use Google trace to test LPST's performance on common file access requests - each task only has one source and one destination. In addition, Google trace provides each task's source machine and starting time but does not describe the data size, network topology, network metrics and destination machine for each task[22], [23]. To make simulation parameters consistent, we use 64MB chunks, 500 and 1500 Mbps network capacity, 10 deadline factor as parameters. In each simulation iteration, we randomly select 30 machines from Google trace data and use 20000 tasks' starting time from these chosen machines for simulation. LPST still performs well. Fig. 4 shows cumulative distribution of algorithms' normalized completion time by using 20000 Google trace tasks' information. Task completion time is normalized by deadline. For 20000 Google trace tasks, LPST completes 95 percent of tasks, and most of them are completed between 0.5 and 0.8 of their deadlines. LPAll finishes 70 percent of tasks; DisFIFO and DisEDF finish 30-40 percent tasks; while FIFO and EDF only finish 5 percent tasks. Although DisFIFO and DisEDF complete more tasks than LPST before 0.5 $\times$ deadlines

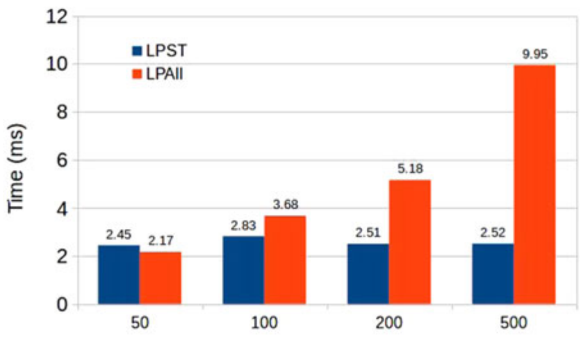


Fig. 5. Scalability of LPST and LPAll with respect to number of tasks.

and LPAll completes more tasks than LPST before 0.7 × deadlines, they complete fewer tasks as expense.

5.6 Overhead

It is quite clear that LPST is more complex than greedy heuristics. But LPST is designed in a practical and scalable way to handle large number of tasks. In this section, we compare the computation cost of LPST with LPAll to evaluate the scalability of LPST. For scalability experiments, we vary the number of tasks and measure the time required to generate a scheduling plan. The results are shown in Fig. 5. We see that LPST's computation time stays roughly the same even if we increase the number of tasks significantly. The computation time of LPAll, however, increases dramatically with the number of tasks. It is mainly because LPST selects only a fixed number of most "emergent" tasks—rather than selecting all tasks as done by LPAll—for computing linear programming functions.

6 Conclusions

In this paper, we consider the problem of optimizing background traffic in erasure-coded distributed storage systems. Our goal is to maximize the number of tasks meeting deadlines under data placement, network topology and bandwidth constraints. The proposed solution makes use of *Remaining Time Flexibility* to select active tasks for each scheduling interval and linear programming to apportion bandwidth among them. Our evaluation results based on both simulations and experiments on a real cluster showed that our proposed algorithm significantly outperforms six competing algorithms. In the future, we plan to evaluate LPST using other topologies, such as fat-tree or Bcube, and prove a performance bound for the algorithm.

References

[1] Ceph, [Online]. Available: http://ceph.com/

[2] OpenStack Object Storage (Swift), [Online]. Available: http://swift.openstack.org

[3] Erasure Coding Support inside HDFS, [Online]. Available: https://issues.apache.org/jira/browse/HDFS-7285

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, et al., "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 143–157.

[5] White Paper: EMC Atmos Cloud Storage Architecture, [Online]. Available: https://www.emc.com/collateral/software/white-papers/h9505-emc-atmos-archit-wp.pdf

[6] Yahoo Cloud Object Store, [Online]. Available: http://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at

[7] Google Colossus File System, [Online]. Available: http://static.googleusercontent.com/media/research.google.com/en/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf

[8] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al., "F4: Facebooks warm blob storage system," in *OSDI*, 2014, pp. 383–398.

[9] Hakim Weatherspoon and John D. Kubiatowicz, "Erasure coding versus replication: A quantitative comparison," in *Proc. Peer-to-Peer Syst.*, 2002, pp. 328–337.

[10] Wallace, Grant, et al., "Characteristics of backup workloads in production systems," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 4–4.

[11] W. Bolosky, J. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop PCS," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, 2000, pp. 34–43.

[12] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, 2013, pp. 325–336.

[13] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, Art. no. 7.

[14] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global EDF scheduling for parallel real-time tasks," *Real-Time Syst.*, vol. 51, no. 4, pp. 395–439, 2015.

[15] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," *Real-Time Syst.*, vol. 49, no. 4, pp. 404–435, 2013.

[16] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proc. 31st IEEE Real-Time Syst. Symp.*, 2010, pp. 259–268.

[17] J. A. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Berlin, Germany: Springer, 2012, vol. 460.

[18] K. A.-S. Al-Harbi, S. Z. Selim, and M. Al-Sinan, "A multiobjective linear program for scheduling repetitive projects," *Cost Eng.*, vol. 38, no. 12, p. 41, 1996.

[19] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 351–362, 2013.

[20] D. Pan and Y. Yang, "FIFO-based multicast scheduling algorithm for virtual output queued packet switches," *Proc. IEEE Trans. Comput.*, vol. 54, no. 10, pp. 1283–1297, 2005.

[21] K. Kogan, A. Lopez-Ortiz, S. I. Nikolenko, and A. V. Sirotkin, "Online scheduling fifo policies with admission and push-out," *Theory Comput. Syst.*, vol. 58, pp. 322–344, 2016.

[22] J. Wilkes, "More Google cluster data," Google research blog, Nov. 2011. [Online]. Available: http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html

[23] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: Format + schema," Google Inc., Mountain View, CA, USA., Nov. 2011. [Online]. Available: https://github.com/google/cluster-data

[24] P. R. Kumar and S. P. Meynf, "Duality and linear programs for stability and performance analysis of queueing networks and scheduling policies," *IEEE Trans. Autom. Control*, vol. 41, no. 1, pp. 4–17, Jan. 1996.

[25] J. Y.-T. Leung, "A new algorithm for scheduling periodic, real-time tasks," *Algorithmica*, vol. 4, no. 1–4, pp. 209–219, 1989.

[26] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 501–521.

[27] L. G. Khachiyan, "Polynomial algorithms in linear programming," *USSR Comput. Math. Math. Phys.*, vol. 20, no. 1, pp. 53–72, 1980.

[28] N. Karmarkar, "A new polynomial-time algorithm for linear programming," in *Proc. 16th Annu. ACM Symp. Theory Comput.*, 1984, pp. 302–311.

[29] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *J. ACM*, vol. 31, no. 1, pp. 114–127, 1984.

[30] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: Novel erasure codes for big data," in *Proc. Int. Conf. Very Large Data Bases*, 2013, pp. 325–336.

[31] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al., "Erasure coding in windows azure storage," in *Proc. Usenix Annu. Tech. Conf.*, 2012, pp. 15–26.
[32] A. G. Dimakis, P. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.
[33] R. E. Tarjan and A. E. Trojanowski, "Finding a maximum independent set," *SIAM J. Comput.*, vol. 6, no. 3, pp. 537–546, 1977.
[34] T. A. Feo, M. G. Resende, and S. H. Smith, "A greedy randomized adaptive search procedure for maximum independent set," *Operations Res.*, vol. 42, no. 5, pp. 860–878, 1994.
[35] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Design Implementation*, 2006, pp. 307–320.
[36] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.
[37] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell, "Ambry: Linkedins scalable geo-distributed object store," in *Proc. 2016 Int. Conf. Management Data*, ACM, 2016, pp. 253–265.
[38] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: A high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 63–74, 2009.
[39] J. W. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint VM placement and routing for data center traffic engineering," in *Proc. IEEE INFOCOM*, 2012, pp. 2876–2880.
[40] C. Koufogiannakis and N. E. Young, "A nearly linear-time ptas for explicit fractional packing and covering linear programs," *Algorithmica*, vol. 70, no. 4, pp. 648–674, 2014.
[41] OpenStack: Open Source Software for Creating Private and Public Clouds, [Online]. Available: http://www.openstack.org/
[42] rsync, [Online]. Available: https://rsync.samba.org/
[43] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endowment*, vol. 6, pp. 1092–1101, 2013.
[44] M. Xia, M. Saxena, M. Blaum, and D. Pease, "A tale of two erasure codes in HDFS" in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 213–226.

Shijing Li received the BE degree from the Beijing University of Posts and Telecommunications, in 2014. She is working toward the PhD degree at the George Washington University. Her research focus includes distributed storage systems, traffic scheduling, and edge computing.

Tian Lan received the BASc degree from the Tsinghua University, China in 2003, the MASc degree from the University of Toronto, Canada, in 2005, and the PhD degree from the Princeton University, in 2010. He is currently an associate professor in electrical and computer engineering with the George Washington University. His research interests include cloud resource optimization, mobile networking, storage systems and cyber security. He received the 2008 IEEE Signal Processing Society Best Paper Award, the 2009 IEEE GLOBECOM Best Paper Award, and the 2012 INFOCOM Best Paper Award.

Moo-Ryong Ra received the PhD degree from Computer Science Department, University of Southern California. He is a principal inventive scientist with the AT&T Labs Research. He is interested in systems and networking. In AT&T, more focus is given to the following areas—software-defined storage for cloud platform/infrastructure in the context of AT&T integrated cloud and edge cloud, video storage and delivery, RDMA networking for next generation storage/memory architecture. Before joining AT&T, he had built several interesting cloud-enabled mobile sensing systems to better understand the interaction between smart mobile devices and the cloud infrastructure. Contact him at mra@research.att.com.

Panta Krishna Rejesh received the PhD degree in electrical and computer engineering from Purdue University. He is a principal inventive scientist with the AT & T Labs-Research. His research interests include cloud computing, Big Data, storage systems, distributed systems, wireless networks, mobile systems, and sensor networks.