FlowTime: Dynamic Scheduling of Deadline-Aware Workflows and Ad-hoc Jobs

Zhiming Hu, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
Email: zhiming@ece.utoronto.ca, bli@ece.toronto.edu

Chen Chen, Xiaodi Ke
Huawei Canada Research Center
Toronto, Canada
Email: {chen.cc, xiaodi.ke}@huawei.com

***Abstract*—With rapidly increasing volumes of data to be processed in modern data analytics, it is commonplace to run multiple data processing jobs with inter-job dependencies in a datacenter cluster, typically as recurring data processing workloads. Such a *group* of inter-dependent data analytic jobs is referred to as a *workflow*, and may have a deadline due to its mission-critical nature. In contrast, non-recurring ad-hoc jobs are typically best-effort in nature, and rather than meeting deadlines, it is desirable to minimize their average job turnaround time.**

The state-of-the-art scheduling mechanisms focused on meeting deadlines for individual jobs only, and are oblivious to workflow deadlines. In this paper, we present *FlowTime*, a new system framework designed to make scheduling decisions for workflows so that their deadlines are met, while simultaneously optimizing the performance of ad-hoc jobs. To achieve this objective, we first adopt a divide-and-conquer strategy to transform the problem of workflow scheduling to a deadline-aware job scheduling problem, and then design an efficient algorithm that tackles the scheduling problem with both deadline-aware jobs and ad-hoc jobs by solving its corresponding optimization problem directly using a linear program solver. Our experimental results have clearly demonstrated that *FlowTime* achieves the lowest deadline-miss rates for deadline-aware workflows and 2-10 times shorter average job turnaround time, as compared to the state-of-the-art scheduling algorithms.

***Index Terms*—workflow scheduling, big data processing, deadline-aware scheduling**

I. INTRODUCTION

Due to their growing complexities, modern commercial applications are commonly represented as a group of inter-dependent Hadoop [1] or Spark [2] jobs [3]. Such a group of inter-dependent jobs is referred to as a *workflow*, and may be associated with a deadline due to the mission-critical nature of these commercial applications. These deadline-aware workflows are typically recurring, running on a daily, weekly or monthly basis. As a consequence, we have rather complete knowledge of each workflow, including its direct acyclic graph (DAG) that represents inter-job dependencies, the resource demand for each job in the workflow, as well as the estimated running time of tasks in each job. Such information will be instrumental for designing new workflow-aware scheduling algorithms that seek to meet workflow deadlines.

It is common for these mission-critical workflows to share the datacenter cluster with ad-hoc jobs, which are best effort and non-recurring in nature [4], [5], [6], without any *a priori* knowledge of resource demands or running time estimates. We will still wish to minimize their job turnaround time, defined as the time of completion minus the time of submission, as we meet deadlines for recurring workflows.

Existing work in the literature does not consider dependencies across jobs [4], [5], [6] or the performance of ad-hoc jobs [3]. Rayon [4], for example, assumed that the deadline for each job is known, which is not the case when deadlines are associated with workflows rather than individual jobs. To resolve this issue, Morpheus [5] proposed to infer the deadlines of jobs from prior runs of workflows. However, it has not utilized global information of the entire workflow, such as how jobs depend upon each other. Li *et al.* [3] ignored ad-hoc jobs, which can be severely delayed by deadline-aware workflows.

In this paper, we argue that deadline-aware workflows and latency-sensitive ad-hoc jobs should be jointly optimized. We present the design and implementation of a new system framework, *FlowTime*, to meet as many deadlines for deadline-aware workflows as possible, and to optimize the average job turnaround time of ad-hoc jobs at the same time. To achieve this objective, *FlowTime* first decomposes the deadlines of workflows to estimated deadlines of their constituent jobs, based on the direct acyclic graph (DAG) within each workflow, used to represent inter-job dependencies.

After deadlines for individual jobs in a workflow have been estimated, *FlowTime* is designed to solve an optimization problem that is specifically formulated to meet workflow deadlines and optimize the average job turnaround time at the same time. Just like typical optimization problems related to resource scheduling [4], [6], our optimization problem is in the category of integer programming problems. The highlight of *FlowTime* is that our specific problem is formulated in such a way that it can be directly solved using a linear program (LP) solver, which we are able to prove theoretically. This way, *FlowTime* schedules resources in a theoretically sound and optimal fashion, which traditional resource scheduling heuristics [4], [5] may not enjoy.

To demonstrate its performance and resource efficiency, we have deployed a real-world implementation of *FlowTime* in YARN, and conducted extensive experiments with standard benchmarks and trace-driven simulations. Our experimental results have clearly shown that *FlowTime* achieves the lowest deadline miss rates while reducing the average job turnaround time of ad-hoc jobs by 2-10 times at the same time.

II. BACKGROUND AND MOTIVATION

In this section, we will briefly talk about the system model and the motivations of our system.

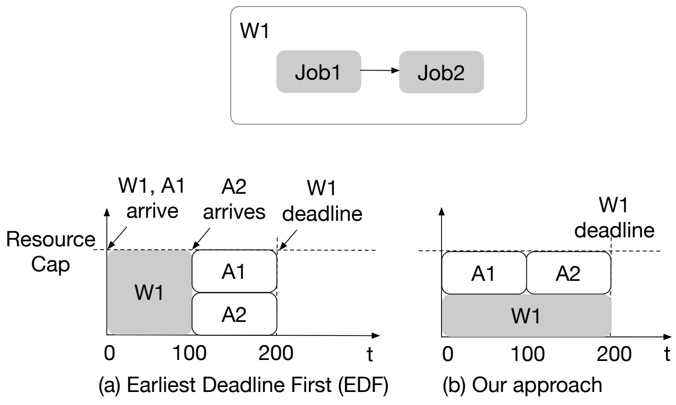


Fig. 1. A motivating example.

A. System Model

We consider two kinds of workloads, which are deadline-aware workflows and latency sensitive ad-hoc jobs. The deadline-aware workflows can be represented $\mathcal{W} = \{W_1, W_2, \cdots, W_i, \cdots, W_l\}$ where l is the total number of workflows. Each workflow has its own starting time and deadline. We use ws_i and wd_i to represent the starting time and deadline of the i-th workflow. Note that, workflows are recurring. Therefore, we know the whole direct acyclic graph (DAG) information of the workflow and all the job information in the DAG within a specified time window.

Without loss of generality, in the i-th workflow, there are m interdependent jobs, which can be represented as $\mathcal{Q}_i = \{u_1, \cdots, u_m\}$. All the jobs that depend on the j-th job in the i-th workflow can be denoted as the set P_i^j. The dependencies among jobs in the i-th workflow can thus be represented in $\mathcal{P}_i = \{P_i^1, \cdots, P_i^m\}$. Therefore, each workflow can be denoted as $W_i = \{\mathcal{Q}_i, ws_i, wd_i, \mathcal{P}_i\}$.

Besides the deadline-aware workflows, the latency-sensitive ad-hoc jobs are also sharing the cluster [5]. These ad-hoc jobs can be submitted to the system at any time. More importantly, they do not have deadlines. Instead, minimizing the average job turnaround time, calculated by the time of completion minus the time of submission, is their ultimate goal. As the ad-hoc jobs are new to the system, the job size information is not available at the time of scheduling. Therefore, the optimization goal of FlowTime is to meet as many deadlines of workflows as possible while minimizing the average job turnaround time of the ad-hoc jobs at the same time.

B. Motivation

Earliest deadline first (EDF) is a naive approach for the problem. However, this approach may block the ad-hoc jobs as long as there are deadline-aware workflows in the cluster even though their deadlines may be far away. As a result, it will incur high job turnaround time for ad-hoc jobs.

To better illustrate this case, we show a motivating example in Fig. 1. In this example, there is one workflow W1, which consists of two interdependent jobs. Along with the workflow, there are two ad-hoc jobs A1 and A2. The arrival time of W1, A1 and A2 are 0, 0, and 100, respectively. The deadline of W1 is 200. Given the earliest deadline first (EDF) approach, W1 will be scheduled first, followed by A1 and A2. As we can see in Fig. 1 (a), A1 is delayed by 100 time units before it can start. The situation could be even worse if there are more deadline-aware workflows in the cluster.

The key reason for the inferior performance of EDF is that it tries to complete the workflows as soon as possible even though their deadlines are pretty loose, which is very common based on our studies on the traces. For instance, in one of our traces, the deadline for the workflow is 24 hours, which is set by its business logic. However, it can complete in only around 2 hours. To avoid this disadvantage, we schedule the deadline-aware workflows while minimally impacting the performance of ad-hoc jobs. In other words, we will schedule the deadline-aware workflows to meet their deadlines while minimizing the max resource consumption in the cluster across time. After that, the remaining resources can be used by the ad-hoc jobs that may arrive at any time. The scheduling results of our approach is shown in Fig. 1 (b). As we can see, we can meet the deadline of W1 and leave the remaining resources to ad-hoc jobs so that the ad-hoc jobs can also be scheduled as early as possible. The average job turnaround time of the ad-hoc jobs is reduced from 150=(200+100)/2 to 100=(100+100)/2.

III. System Design

In this section, we will present the overview of the design of FlowTime.

A. Desired Features

Before we introduce the design, here we first show the desired properties of our scheduling system, which will be considered throughout of our design.

Scheduling quality: The quality of the scheduling is always our top concern. In our case, the quality means that more deadlines of workflows are met or lower average job turnaround time of ad-hoc jobs is achieved.

Scheduling efficiency: Besides the scheduling quality, scheduling efficiency is another critical factor to be considered. As the job scheduling system runs in a dynamic environment, an ideal scheduler should be able to react to the task/job completion events efficiently. In this sense, our scheduler should be efficient and scalable to the number of jobs in the system and delivers the scheduling results within a limited time.

Robustness to estimation errors: Even though the deadline-aware workflows (jobs) are recurring, the input data or the code may have changed in different runs of the same jobs, which will lead to estimation errors as the estimations are conducted based on the data in prior runs of the jobs. Both underestimations or overestimations are possible in the real cases. The scheduling system should be able to handle both types of estimation errors.

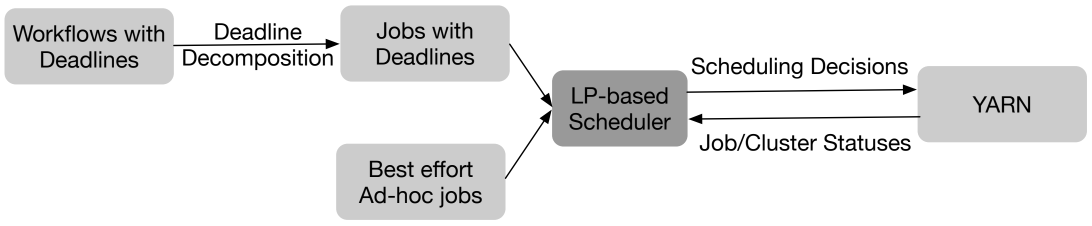


Fig. 2. The overview of FlowTime.

B. Design Overview

The overview of FlowTime is shown in Fig. 2. We can see that the first step is deadline decomposition where the deadline of a workflow will be decomposed into the deadlines of jobs in that workflow. After that, the jobs with deadlines along with ad-hoc jobs will be submitted to the LP-based scheduler, which is the core module for our scheduling system. The LP-based scheduler will directly interact with the cluster resource manager YARN in two ways. On one hand, it dynamically generates scheduling decisions and applies them in YARN. On the other hand, it will keep track of the job statuses and receive the job updates such as task completions or job completions from YARN.

Deadline Decomposition: The deadline decomposition is the first module of our system. The reasons for deadline decomposition are two-fold. First, rather than scheduling the workflows as soon as possible, we can choose to try to finish the workflows just before their deadlines. Therefore, we can decompose the deadlines of workflows and take the deadlines of jobs as milestones for completing the whole workflow. Second, directly modeling the dependencies among jobs and scheduling the workflow is an intractable optimization problem. Therefore, we propose to decompose the deadlines of workflows and take the deadlines of jobs to guarantee the dependencies across jobs in a workflow. Moreover, the problem can be fit in our efficient optimization framework after decomposition.

In our design, the deadlines of workflows will be decomposed into the deadlines of jobs. More specifically, we propose an efficient deadline decomposition algorithm whose complexity is linear with the number of nodes and edges of the directed acyclic graph (DAG) of the workflow. The outputs of the deadline decomposition will be fed to the LP-based scheduler.

LP-based Scheduler: After the deadline decomposition, jobs will be scheduled dynamically by solving a linear programming (LP) problem. In this way, we can regenerate the scheduling results once the job/cluster status changes such as when a task/job completes. As the big data processing applications are prone to estimation errors, our dynamic scheduling strategy is more flexible and robust to the estimation errors.

A key insight for the joint optimization of deadline-aware jobs and ad-hoc jobs is that we schedule the deadline-aware jobs while *minimally* impacting the performance of ad-hoc jobs. To achieve this, the deadline-aware jobs are scheduled while minimizing the max resource utilization in the cluster so that the ad-hoc jobs can be scheduled as soon as possible because they may arrive at any time. However, if there are no ad-hoc jobs, the remaining resources will be reallocated to the deadline-aware jobs to achieve *work conservation*. More details are followed in Sec. V.

IV. Decompose the Deadlines of Workflows

In this section, we will talk about how we decompose the deadlines of workflows into the deadlines of jobs. This step transforms the complex DAG scheduling problem into a simpler problem, which can be further efficiently solved by an LP solver.

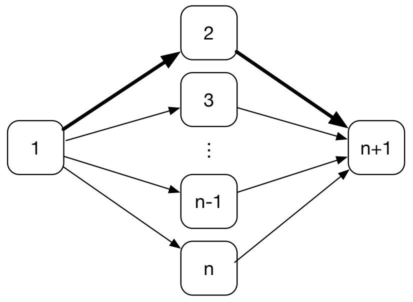
There are a few key requirements in the design. First, the duration allocated for each job should be larger than the minimum time needed for each job. For instance, for a MapReduce job, if the estimated running time of map tasks and reduces are five seconds and four seconds, then the minimum running time of the job is nine seconds. Second, the durations for different jobs should not violate the dependencies among different jobs in each workflow. For instance, if job 2 depends on job 1, then the starting time of job 2 should not be earlier than the deadline of job 1. Finally, the deadline decomposition algorithm should consider that the resources in a cluster are limited. Otherwise, unreasonable deadline assignments may lead to infeasible scheduling of jobs in the cluster.

Based on the principles of design, we propose a two-step approach for the deadline composition problem. Given a DAG, we first obtain a sequence of node sets where dependencies only exist among different sets. After we obtain such sequence, we then distribute the total deadline to these node sets based on the total resource demands in each node set. For instance, the sequence of the three node sets in Fig. 3 is $\{1, \{2, 3, \cdots, n\}, n + 1\}$.

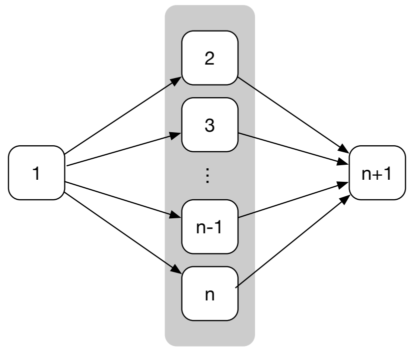
A. A Variant Topological Order

This step is designed to meet the second and third requirements. As the nodes in the same node set do not have dependencies with each other, they can run in parallel and thus share a same time window. Moreover, by this way, we can consider the jobs in the same node set altogether in the process of deadline decomposition.

The algorithm is basically about obtaining a topological order for the nodes in a DAG. However, the differences are that we will group the consecutive nodes that have no dependencies



(a) State-of-art approach



(b) Our approach

Fig. 3. The state of the art [7] and our approach.

together. As a result, we can group the nodes that do not have the dependencies together without violating the dependencies. We adopt Kahn's algorithm [8] to obtain the topological order. For instance, the original output for the topological order of the nodes in the DAG in Fig. 3 is $\{1, 2, \cdots, n, n+1\}$ while the output of our algorithm will be $\{1, \{2, 3, \cdots, n\}, n+1\}$.

B. Resource Demand Based Deadline Decomposition

After we have a sequence of node sets, what we need to decide are the deadlines for all the node sets. First of all, we need to guarantee the minimum runtime for each node set, which is decided by the largest minimum runtime for all the jobs in the same node set. Therefore, we first calculate the minimum runtime for all the node sets and allocate the minimum duration to those sets. But the question is, how should we allocate the remaining time to those node sets[1]?

A naive approach could be assigning the remaining deadlines to the node sets based on their minimum runtime. However, this approach does not consider the amount of resources that are needed in each node set. Therefore, if there are too many parallel jobs in a node set assigned with the same deadlines, they may not be able to meet their deadlines as the resources of clusters are limited. Instead, we argue that the deadlines distributed to different node sets should take the resource demands of the whole set into consideration. To this end, we propose to allocate the remaining deadlines to the node sets based on the total resource demands in the each node set where the resource demands are calculated according to the number of tasks, the task running time and the resource requirement of each task.

[1]In some cases, the remaining time is negative. We will use the critical path based approach in [7] to decompose the deadlines of workflows instead.

TABLE I
NOTATIONS USED IN THIS PAPER.

Symbol	Meaning
x_{it}^r	The amount of type r resource allocated to job i at time slot t
z_t^r	The total amount of type r resource allocated at time slot t
C_t^r	The total amount of type r resource at time slot t in the cluster
a_i	The arrival time of job i
d_i	The deadline of job i
s_i^r	The resource need of type r resource for job i

We have a simple example in Fig. 3. In this figure, there are (n+1) jobs in total and we assume the starting time of job 1 is 0. In our case, we also assume all these jobs have the same running time and the same resource demands. As the jobs from job 2 to job n are parallel jobs , they will be assigned the same arrival time and deadlines. In the traditional approach [7], it will first find a critical path based on the running time of the jobs in the graph and decompose the deadlines based on the runtime of the jobs in the critical path. For instance, in this case, 1–>2–>(n+1) is the critical path and the job 2 will get 1/3 of the total deadline, which is also the deadline for all the parallel jobs in the middle of the graph. However, the traditional approach ignores resource demands of the jobs, which is problematic as the cluster resources are limited and there may not be enough resources to host all those parallel jobs if n is very large. For instance, if the amount of resources in the cluster cannot support all the parallel jobs from job 2 to job n in the same duration, it will be infeasible to place all those jobs in this allocated duration.

Instead, in our approach, we both consider the job running time and the resource demand of the jobs in each node set when decomposing the deadlines. More specifically, we take all the (n-1) jobs in the middle into consideration at the same time as they need more time to complete in a resource limited cluster. In other words, we consider all the parallel jobs together when allocating the deadlines. Therefore, the deadline assigned to job 2 to job n will be (n-1)/(n+1) of the total deadline in our approach instead of 1/3 of the deadline in the traditional approach.

V. Dynamic Scheduling with an LP

After the deadline decomposition, now we have transformed the problem from scheduling the deadline-aware workflows to scheduling deadline-aware jobs and latency-sensitive ad-hoc jobs. In this section, we will show how we schedule these two types of jobs efficiently while achieving their separate goals.

A. The Original Formulation

After the deadlines of jobs are decided, we are now ready to allocate the resources for the jobs including deadline-aware jobs and ad-hoc jobs. More specifically, we propose to allocate the resources to the deadline-aware jobs and guarantee their deadlines while minimally impacting the performance of ad-hoc jobs. We adopt a slot-based formulation[2] and he problem can be formulated as below. The meanings of the notations are shown in Table I.

$$\begin{aligned}
&\underset{\boldsymbol{x},\boldsymbol{z}}{\text{lexmin}} && \max \quad z_t^r/C_t^r && (1)\\
&\text{s.t.} && \sum_{t=a_i}^{d_i} x_{it}^r = s_i^r, \quad \forall i \in \mathcal{N}, \forall r \in \mathcal{R} && (2)\\
& && \sum_{i=1}^{n} x_{it}^r = z_t^r, \quad \forall t \in \mathcal{T}, \forall r \in \mathcal{R} && (3)\\
& && z_t^r \le C_t^r, \quad \forall t \in \mathcal{T}, \forall r \in \mathcal{R} && (4)\\
& && x_{it}^r \in N^0. \quad \forall i \in \mathcal{N},\ \forall t \in \mathcal{T}, \forall r \in \mathcal{R} && (5)
\end{aligned}$$

This formulation only includes deadline-aware jobs and the goal is to minimize the max resource usages after placing the deadline-aware jobs so that the ad-hoc jobs can be scheduled as early as possible to reduce the average job turnaround time. Therefore, we can meet the deadlines of deadline-aware jobs and reduce the average job turnaround time of ad-hoc jobs at the same time.

In this formulation, x_{it}^r denotes the amount of type r resource that will be allocated to the i-th deadline-aware job at the t-th time slot. We can see that we first accumulate the total amount of resource allocated to all the n deadline-aware jobs and normalized it with the total amount of type r resource in the cluster at time slot t. The normalization is because we want to make it comparable for different types of resources. After normalization, we aim to obtain the lexicographical minimal vector over different r and t. In other words, we prefer more balanced allocations across different time slots and resource types.

The first constraint in Eq. (2) shows that we need to satisfy the resource requirements of each job for all the resource types from its arrival time to its deadline. Here s_i^r represents the amount of type r resource is needed for the i-th job. a_i and d_i are the arrival time and deadline of i-th job.

The second constraint in Eq. (3) shows that the total amount of resource used by all the n deadline-aware jobs is denoted by z_t^r for all the time slots and all the resource types.

The third constraint in Eq. (4) means that the total amount of resources allocated to all the deadline-aware jobs should not exceed the total amount of resources available in the cluster where C_t^r is the resource cap for the type r resource at t-th time slot. In our case, the resource cap could vary with time to provide more flexibility to different situations.

In the last constraint, x_{it}^r can only be nonnegative integers. This is because in some resources, we can only use integers to represent the amount. For instance, in YARN [9], the number of CPU cores that will be allocated to the application has to be integers. Therefore, we use integers to represent the resource allocations. Similar settings are also used in [10].

[2]The duration of one slot is discussed in Sec. VI.

As we can see, the original problem can be formulated as an integer linear programming (ILP) problem, which normally cannot be solved efficiently. Fortunately, we find out that we can transform the original ILP problem and reduce it to a linear programming (LP) problem as shown in the following section.

B. The Equivalent LP

The sort of integer linear programming (ILP) problems that can be transformed into linear programming problems should meet two conditions [11]. First, the objective function should be a separable convex objective function, which means that the objective function is separable and each part is convex. The other condition is that the constraint matrix formed by the coefficients of the constraints should be a totally unimodular matrix.

Therefore, we first prove that the lexicographical minimal vector can be achieved by minimizing a scalar instead in **Lemma** 1 where $\boldsymbol{u} \preceq \boldsymbol{v}$ means that vector $\boldsymbol{u}$ is lexicographical no greater than $\boldsymbol{v}$. We use

$$g(\boldsymbol{u}) = \sum_{i=1}^{k} k^{u_i}, \tag{6}$$

to transform the vector to a scalar and k is the dimension of the vector. Therefore, $k = |\mathcal{T}||\mathcal{R}|$ in our case.

Lemma 1. *For $\boldsymbol{u},\ \boldsymbol{v} \in \mathbb{Z}^k,\ g(\boldsymbol{u}) \le g(\boldsymbol{v}) \iff \boldsymbol{u} \preceq \boldsymbol{v}$.*

Given **Lemma** 1, we can transform the original objective function to the new objective function as shown below. We can clearly see that the new objective function can be separated into multiple convex functions, which meets the first condition.

$$\underset{\boldsymbol{x},\boldsymbol{z}}{\text{lexmin}} \ \max \ z_t^r/C_t^r = \min \ \sum_{t\in\mathcal{T}}\sum_{r\in\mathcal{R}} k^{z_t^r/C_t^r} \tag{7}$$

We further prove that the coefficients in the constraint matrix form a totally unimodular matrix in **Lemma** 2, which is an important indicator about whether an LP has integer solutions. This is because if the constraint matrix is a totally unimodular matrix, then the feasible region is an integral polyhedron and only has integral extreme points. LP solver algorithms like Simplex will search the optimal solutions from one extreme point to another. Consequently, the solutions can be guaranteed to be integral and our problem formulation can also meet the second condition.

Lemma 2. *The coefficients in the constraints (2), (3), (4) and (5) form a totally unimodular matrix.*

After transformation, we can finally make the problem equivalent to a problem that can be solved by efficient LP solvers. Here we can now obtain the equivalent LP using the λ-representation as shown below:

$$f(y) = \sum_{j\in\mathcal{D}} f(j)\lambda_j, \quad \sum_{j\in\mathcal{D}} j\lambda_j = y, \tag{8}$$

$$\sum_{j\in\mathcal{D}} \lambda_j = 1, \quad \forall \lambda_j \in R^+, \forall j \in \mathcal{D}. \tag{9}$$

In the λ-representation, $\mathcal{D}$ is the set that contains all the possible values of y. It introduces an auxiliary variable λ_j for every possible values of y and y can be denoted as the linear combinations of λ_j and the corresponding possible values.

With λ-representation, the above-mentioned integer linear programming (ILP) problem in Eq. (1) can be transformed into the following LP problem:

$$\min_{\boldsymbol{x},\boldsymbol{z},\boldsymbol{\lambda}} \quad \sum_{t\in\mathcal{T}}\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{D}^r} k^{j/C_t^r}\cdot\lambda_{tj}^r \tag{10}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{D}^r} j\lambda_{tj}^r = z_t^r, \quad \forall t\in\mathcal{T},\forall r\in\mathcal{R} \tag{11}$$

$$\sum_{j\in\mathcal{D}^r}\lambda_{tj}^r = 1, \qquad \forall t\in\mathcal{T},\forall r\in\mathcal{R} \tag{12}$$

$$\lambda_{tj}^r\in R^+, \qquad \forall t\in\mathcal{T},\forall r\in\mathcal{R},\forall j\in\mathcal{D}^r \tag{13}$$

$$j\in D^r, \qquad \forall r\in\mathcal{R} \tag{14}$$

$$(2),(3),(4),(5).$$

The detailed proof are similar with the existing work in the literature [12], [13]. We omit the proofs of the lemmas for space limitations. The final LP problem can be efficiently solved by off-the-shelf commercial LP solvers such as CPLEX [14].

VI. Implementations

We implement a plug-in scheduler in YARN based on the capacity scheduler [15]. More specifically, each job will be assigned a unique queue when it is admitted to the system. Therefore, we can dynamically control the amount of resources that will be allocated to the applications by updating the queue configurations in capacity scheduler on a real-time basis. On the other hand, we will also constantly check the application status from YARN [9] to know the remaining resource demands of jobs, which are the inputs for the LP-based scheduler. Our scheduler will be triggered by a task/job completion event or a predefined time limit.

Before the scheduler, we need to prepare the inputs for the LP solver such as the duration of a time slot, remaining number of time slots to the deadline and the remaining resource demands of the jobs. Among which, deciding the duration of each time slot is the key step. In our implementations, we use the greatest common factor (GCF) of the task running time including map task running time and reduce task running time as the duration of one time slot. After that, we then can directly know the number of time slots to the deadline for each job, which is d_i in the formulation. We can also calculate the remaining resource demands s_i^r based on task running time and the duration of each time slot. For example, if a job only has four tasks left and the running time of tasks are both 10 seconds, the remaining resource demand should be 4*10/5 if the duration of each time slot is five seconds. Now we have all the required inputs for the LP scheduler. We can run it and obtain the scheduling results for deadline-aware jobs. The remaining resources will be allocated the ad-hoc jobs if there is any. Otherwise, the remaining resources will be redistributed to the deadline-aware jobs using the earliest deadline first (EDF) strategy to achieve *work conservation*.

In some cases, it is impossible to schedule all the pending jobs by their deadlines given the limited resources in the cluster. In these cases, there will be no feasible solutions for the LP-based scheduler. Therefore, we propose to schedule the jobs from the workflows with the earliest deadline first. This design is the key for the principle of workflow-aware, which can effectively avoid the cases when most workflows complete most of the jobs while still miss the deadlines.

We also find out that the *deadline slack*, which is the gap between the real deadline and the deadline we use in the LP, can further reduce the deadline miss rates. For instance, if the real deadline of the job is 100 seconds and the *deadline slack* is set to be 20 seconds, then the deadline we use in the LP-based scheduler is 80 seconds. The intuition behind the design of *deadline slack* is that we may allocate the resources to the job at the very last minute and we may miss the deadline if the job takes longer than expected. However, with deadline slack, we will try to allocate enough resources to the jobs ahead of the real deadlines and we still have the chance to meet the real deadline even if it takes longer than expected. We will show the performance of *deadline slack* in the evaluations.

VII. Evaluations

In this section, we show the evaluation results of our approach compared with other algorithms.

A. Experimental Setup

Here we show our experimental setups in the following aspects.

Cluster configurations: We start 15 VM instances, each with 4 CPU cores and 15 GB of main memory, in Google Compute Engine. One instance is the master node and the remaining 14 nodes are the slave nodes. In each node, 12 GB of main memory is configured as the memory used by the node manager. Therefore, we can start up to 6 containers in each node and 84 containers in the whole cluster.

We use Hadoop Distributed File System (HDFS) as the underlying file system and the block size is 128 MB. The HDFS shares the master node and slave nodes with YARN. The number of replicas is set to be 3.

Workflows and jobs: We have 5 workflows with 90 jobs in total and different deadlines. On top of that, we have 10 ad-hoc jobs with random arrival time and job types. Each workflow is formed by 18 MapReduce jobs with the LIGO topology [16], which is widely used in workflow related studies. We replace the jobs in the topology with MapReduce jobs and we use the same MapReduce jobs if the nodes are representing the same task types in the original topology.

The MapReduce jobs are selected from the following eight jobs, which are Classification, HistogramMovie, HistogramRatings, InvertedIndex, SequenceCount, WordCount, SelfJoin and TeraGen in the PUMA benchmark [17]. For Classification, HistogramMovie and HistogramRatings, we adopt the movies dataset in the benchmark. Wikipedia datasets are used for word processing related applications, which are InvertedIndex, Sequence-Count and WordCount. SelfJoin takes the generated the datasets as the input.

The input sizes of jobs are at least 10 GB and we process more than 1 TB of data in each round of experiments. Each round takes more than 3 hours to complete in our cluster.

Baselines: We compare our approach named FlowTime with Morpheus [5], CORA [10], FAIR, FIFO and earliest deadline first (EDF). All these baselines only care about the job level performance. To achieve a fair comparison, we consider two types of job in CORA, which are deadline-critical jobs and deadline-sensitive jobs. The default utility functions are used for these two types of jobs.

Metrics: Our metrics are the number of jobs/workflows that can meet the deadlines and average job turnarounds time of ad-hoc jobs. Besides these metrics, we also evaluate the scalability of our approach for both the deadline composition algorithm and the LP-based scheduler.

B. Experimental Results

In the experiments, we want to answer the following questions. 1) What is the performance of the scheduler regarding meeting the deadlines of jobs and workflows? 2) What is the performance of ad-hoc jobs when they are coexisted with deadline-aware workflows and jobs? 3) Is the solution scalable? What is the running time of the algorithms in the system? Below, we will try to answer the questions one by one.

1) Deadline-Aware Jobs/Workflows and Ad-hoc Jobs: In Fig. 4, we show the differences between the completion time and deadlines, the number of jobs missed the deadlines and the average job turnaround time of ad-hoc jobs.

More specifically, in Fig. 4(a), we can see that FlowTime performs the best and all the jobs finish before the deadlines. However, for the other four algorithms, quite a lot of jobs missed the deadlines. Especially, for Fair and FIFO, their performance are the worst because they does not care about the deadlines in the algorithm. EDF is the best among the baselines as it sacrifices the performance of ad-hoc jobs and always schedules the deadline-aware jobs first. However, the performance of EDF is still worse than our algorithm because EDF schedules the jobs one by one, which cannot fully utilize the cluster. CORA considers the performance of the two kinds of jobs. However, the ultimate objective is to minimize the max utilities of jobs instead of maximize the number of jobs that can meet the deadlines or minimizing the average job turnaround time of ad-hoc jobs. Therefore, CORA can only obtain a moderate performance across the baselines.

We can further validate the performance of the algorithms in Fig. 4(b). In this figure, we can see that we meet all the deadlines of the deadline-aware jobs out of 90 deadline-aware jobs. The number of jobs that miss the deadlines for the baselines are 10, 5, 8, and 13, respectively. The reasons for the performance are as explained above.

Besides the performance of deadline-aware jobs, we also present the results of ad-hoc jobs in Fig. 4(c) where our algorithm greatly outperforms the other four algorithms. More specifically, Fair performs the best among the baselines and we can reduce the average job turnaround time by 36%. For other algorithms, our performance is way better. For instance, the average job turnaround time is 1/2 of CORA, 1/3 of FIFO and 1/10 of EDF. Again, we can see that EDF trades the performances of ad-hoc jobs for better performance of deadline-aware jobs. Therefore, it can obtain the fair performance for deadline-aware jobs while receiving poor performance for ad-hoc jobs. However, our solution can jointly optimize these two kinds of jobs and obtain a good performance for both of the jobs. The benefits originate from the fact that we always schedule the deadline-aware jobs while minimally impacting the performance of ad-hoc jobs.

Besides the number of jobs that meet the deadlines, the number of workflows that can meet the deadlines is also an important metric to our scheduling system. As we run a total of 90 deadline-aware jobs, which only contain 5 workflows with each workflow consisting of 18 jobs. The number of workflows that can meet the deadlines are similar. For instance, in the case shown in Fig. 4, the number of workflows that can meet the deadlines are both 5. Even though the performance are similar given the numbers, our algorithm is more predictable as it can meet all the deadlines of jobs, which are the milestones, inside the workflows and is also more friendly to ad-hoc jobs that may arrive at any time with any size.

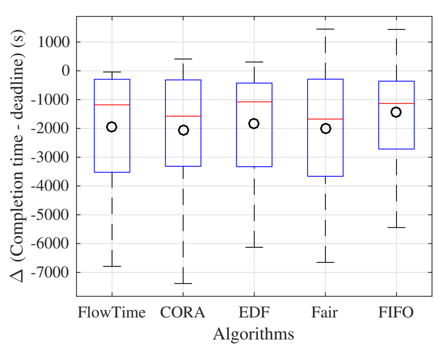
2) The Effectiveness of Deadline Slack: Deadline slack is a very important feature in our system. As we discussed, if we directly use the deadline of the jobs in the formulation as it is, there might be some cases that the jobs will be allocated resources at the very last minute, which can further cause deadline misses. Therefore, here we compare the performance with/without deadline slack in Fig. 5 where FlowTime_no_ds denotes the FlowTime algorithm without the feature of deadline slack.

In Fig. 5(a), we can see that the FlowTime meets the deadlines for all the deadline-aware jobs while FlowTime_no_ds misses some deadlines. We can further verify the results in Fig. 5(b) where 5 jobs miss the deadlines in FlowTime_no_ds. We can find out that with deadline slack, we can reduce the chance of missing the deadlines as we try to meet the resource demands of jobs slightly before the deadlines. In all the figures without explicit statement, the deadline slack is set to be 60 seconds[3].

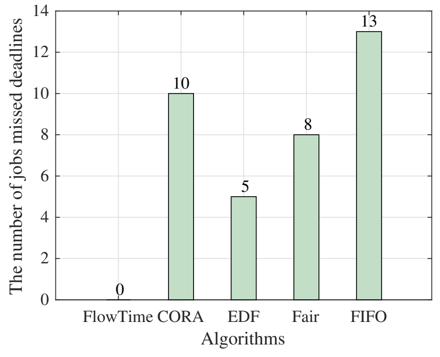
Using deadline slack, we may allocate resources to the jobs earlier than the real deadline, which may affect the performance of ad-hoc jobs. Consequently, we also show the results of the average job turnaround time of ad-hoc jobs in Fig. 5(c). In this figure, we can see that the average job turnaround time are not affected as the we only use a small amount of time for the deadline slack.

3) The Scalability of Deadline Decomposition Algorithm: We also record the running time of deadline decomposition

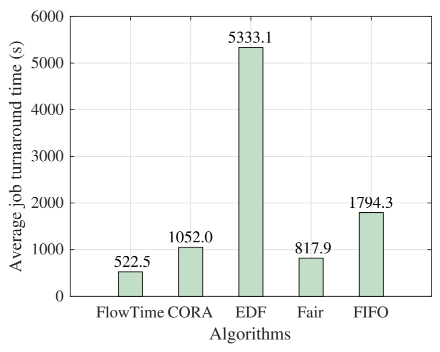
[3]The deadline slack is set empirically. Optimal settings of the deadline slack for different workloads are left for the future work.



(a) Δ (completion time - deadline).

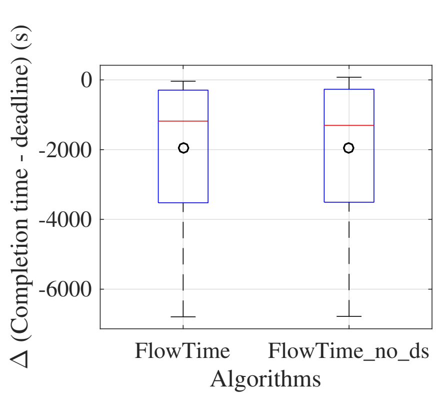


(b) The number of jobs that miss their deadlines.

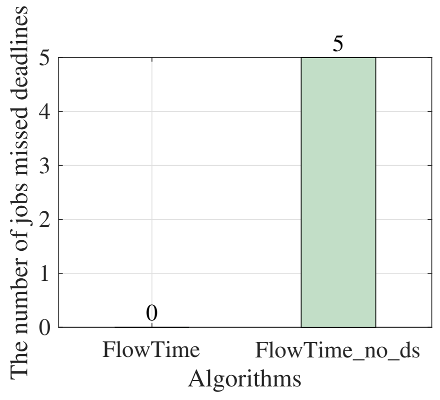


(c) The average job turnaround time of ad-hoc jobs.

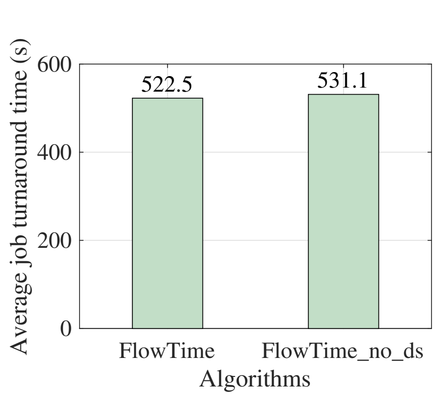
Fig. 4. The performance of the algorithms regarding meeting the deadlines and average job turnaround time.



(a) Δ (completion time - deadline).



(b) The number of jobs that miss their deadlines.



(c) The average job turnaround time of ad-hoc jobs.

Fig. 5. The effects of deadline slack.

algorithm with different number of nodes and edges where the runtime is taken as the average over 1000 runs of the deadline decomposition after 100 warmup runs. The number of nodes ranges from 10 to 200 and we record 5 data points with similar number of edges for each number of nodes. The measurements are conducted on a laptop with Intel Core i7-3630QM 2.4GHz 4-core processor with 8 GB of main memory.

The results are shown in Fig 6. In this figure, we can see that we can efficiently decompose the deadlines of workflows to the deadlines of jobs. The runtime of the algorithm grows slowly with the number of edges and nodes. Even in the case with 200 nodes and 6000 edges, we can still return the results within 3 seconds. Note that, each node is a job, therefore the workflow with 200 jobs is a very big workflow actually. In the reality, we also do not have thousands of edges, which denote dependencies among jobs, in most cases.

4) The Solver Latency: There are mainly two parts in the system, which are deadline decomposition and LP-based scheduler. Hence, besides the efficiency of deadline decomposition algorithm, the LP-based scheduler should also be efficient as it will be triggered whenever a task/job completes. Otherwise, it will incur high scheduling latency and harm the performance of jobs.

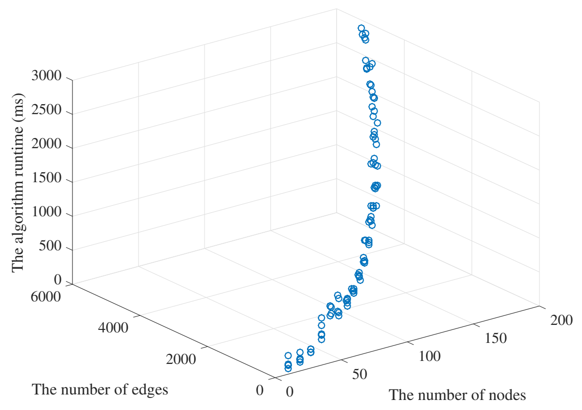


Fig. 6. The runtime of our deadline decomposition algorithm.

We show the running time of the LP-based algorithm in Fig. 7 with the number of deadline-aware jobs. The capacity of the cluster is 500 CPU cores and 1TB of main memory. The number of time slots is set to be 100, which corresponds to the time span of 1000 seconds as the duration for one slot is 10 seconds. We use the CPLEX solver on a MacBook

Pro with 2.6 GHz Intel Core i7 CPU and 16 GB of main memory. The method used in the solver is the network simplex algorithm. Each case is evaluated several times and the runtime is averaged.

We can see that our algorithm can delivery the scheduling results within two seconds for even the large cases. In the figure, we can see that it can return the results within one second when the number of jobs is less than 500 and two seconds when the number of jobs is 900. Even with 1000 jobs, the solver latency is around 2.8 seconds. To note that, it is so efficient because we transformed the original integer linear programming problem to a linear programming problem. The results strongly supports the scalability of our LP-based scheduler.

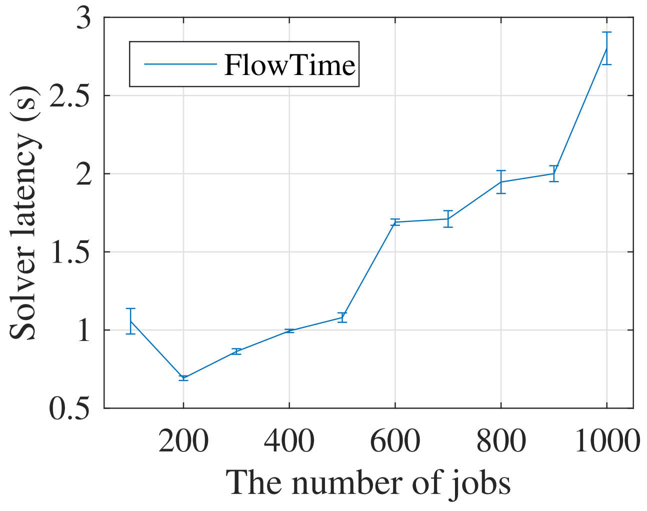


Fig. 7. The running time of the solver.

C. Trace-Driven Simulations

We further compare the performance of our algorithm with Morpheus [5] with a one-day production trace running complex hive SQL queries. In the one day trace, there are 60 workflows and 247 MapReduce jobs. The resource demand of the jobs varies from tens to tens of thousands. In this experiments, we first decompose the deadlines of the workflows to the deadlines of jobs and then compare the packing efficiency of our approach with Morpheus because Morpheus can only handle job-level packing.

Our results is shown in Table II. In this table, we show the normalized allocation area of all the successfully placed jobs. The allocation area for each job is calculated by the product of the number of tasks, the task runtime and the task resource demand. In this cluster, the total amount of memory varies from 320 GB to 390 GB and there is one CPU core for every 2 GB of main memory. We can see that we can place all the jobs with FlowTime with only 320 GB of main memory. However, Morpheus needs 390 GB of memory to place all the jobs. In the case with only 320 GB of main memory, we can allocate 5% more area than Morpheus. The superior performance of our approach is that we pack all the deadline-aware jobs altogether instead of one by one as in Morpheus.

TABLE II
THE NORMALIZED ALLOCATION AREA IN DIFFERENT CLUSTER SIZES.

Total memory (GB)	320	360	370	390
Morpheus	6.997	7.000	7.101	7.257
FlowTime	7.257	7.257	7.257	7.257

VIII. Related Work

In this section, we briefly introduce the most related paper in the scheduling systems for big data processing applications.

The most closely related paper is CORA [10], which was designed to achieve the max-min fairness of utilities for jobs with different utility functions. Therefore, in their paper, the authors also consider both the deadline-critical jobs and deadline-sensitive jobs, which are called deadline-aware jobs and latency-sensitive ad-hoc jobs in our case. The major differences are that we have different scheduling objectives and we do not assume that we know the job details of ad-hoc jobs beforehand, which are closer to the real cases in production clusters [4], [5].

Another paper that worked on workflow scheduling is WOHA [3], which proposed a *progress-based* method to prioritize the workflows dynamically. For this reason, they need to first simulate the executions of every single workflow to obtain the relationship between *ttd* (time to deadline) and the number of tasks that have completed. However, in the simulation, they need to specify the amount of resources that will be allocated to that workflow, which is hard to estimate given that many other workflows are contenting the resources in the same cluster. Inaccurate progress estimation may result in poor performance for meeting the deadlines. In our design, we do not need to specify the relationship between *ttd* and the number of complete tasks. Another difference is that we also aim to improve the performance of ad-hoc jobs, which is another important type of workloads in real clusters [4], [5].

There are some papers that both consider recurring deadline-aware jobs and ad-hoc jobs [4], [5], [6] in the job level. In other words, only the **job level performance** were considered in the paper. Among which, Rayon [4] and Morpheus [5] are reservation based systems, which are orthogonal to our approach. Combining the reservation system and our dynamic scheduling approach is part of our future work. More specifically, Rayon [4] designed a Reservation Definition Language (RDL) and formulated the reservation problem as an Integer Linear Programming (ILP) problem. But a heuristic approach was adopted for practical issues. The basic idea is to schedule the deadline-aware jobs as late as possible to leave room for ad-hoc jobs. In Morpheus [5], the authors first inferred the deadlines of jobs from historical data and then placed the jobs one by one while trying to minimize the max resource utilization of the cluster. The above two papers both focused on the reservation systems. A scheduling oriented approach was proposed in TetriSched [6], which aimed to improve the scheduling performance in heterogeneous environments. All these work only focused on the **job level performance** such as how many jobs meet the deadlines, which leaves potential performance improvements in the **workflow level**.

Besides the scheduling systems for the workloads of big data analytics. Scientific workflow scheduling has been investigated for a long time in grid computing [7] and public clouds [18]. In [7], it proposed to minimize the execution cost for workflows with deadlines. In this paper, a deadline decomposition algorithm is also proposed for simple DAGs. Instead, the algorithm in [18] was designed for public clouds where on-demand resource provisioning and the pay-as-you-go pricing model are considered for calculating the costs. However, both of the above-mentioned papers, their workflows only contain tasks, which have determined running time on a specific type of computing resources. In our case, each node in the workflow is a job and the computation time for the node is undetermined. Moreover, we aim for improving the scheduling performance regarding meeting the deadlines of workflows and reducing the average job turnaround time of ad-hoc jobs instead of minimizing the overall monetary costs.

Job scheduling in geo-distributed big data processing systems were studied in [12], [19], [20], [21], [22], [23], [24], [25], [26]. In [26], the authors proposed a data placement and a reduce task scheduling algorithm to reduce the job completion time of Spark applications across geo-distributed data centers. While the ultimate goal in [22] is to reduce the data transfers across geo-distributed data centers. In the paper, they proposed a variant min-k-cut algorithm to cut the graph to decide the task placements in several data centers. Instead of optimizing job completion time or data transfers, the bandwidth costs incurred by data transferred across geo-distributed data centers were optimized in [25] for SQL query related applications. The above-mentioned approaches only focused on the performance of one job. The case of multiple jobs was discussed in [21] where an efficient heuristic approach was proposed to reduce the average job completion time.

IX. Conclusion

In this paper, we have proposed *FlowTime* to meet the deadlines of workflows and to minimize the average job turnaround time of ad-hoc jobs at the same time. We first design a deadline decomposition algorithm that can efficiently decompose the deadlines of workflows to the deadlines of jobs. We then jointly optimize the performance of deadline-aware jobs and latency-sensitive ad-hoc jobs by scheduling the deadline-aware jobs while minimally impacting the performance of ad-hoc jobs. To achieve this, we have proposed to transform the original integer linear programming (ILP) problem to an efficient equivalent linear programming (LP) problem that can be efficiently solved by standard LP solvers. The experimental and trace-driven simulation results strongly confirm the effectiveness and efficiency of our system.

References

[1] "Hadoop," https://hadoop.apache.org/, accessed: 2017-10-16.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proc. USENIX NSDI*, 2012.

[3] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "Woha: Deadline-Aware Map-Reduce Workflow Scheduling Framework over Hadoop Clusters," in *Proc. IEEE ICDCS*, 2014.

[4] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based Scheduling: If You're Late Don't Blame Us!" in *Proc. ACM SoCC*, 2014.

[5] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards Automated SLOs for Enterprise Clusters." in *Proc. USENIX OSDI*, 2016.

[6] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters," in *Proc. ACM Eurosys*, 2016.

[7] J. Yu, R. Buyya, and C. K. Tham, "Cost-Based Scheduling of Scientific Workflow Applications on Utility Grids," in *In Proc. IEEE e-Science and Grid Computing*, 2005.

[8] A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *Proc. ACM SoCC*, 2013.

[10] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need for Speed: Cora Scheduler for Optimizing Completion-Times in the Cloud," in *Proc. IEEE INFOCOM*, 2015.

[11] R. Meyer, "A Class of Nonlinear Integer Programs Solvable by a Single Linear Program," *SIAM Journal on Control and Optimization*, vol. 15, no. 6, pp. 935–946, 1977.

[12] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling Tasks Closer to Data across Geo-distributed Datacenters," in *Proc. IEEE INFOCOM*, 2016.

[13] ——, "Time- and Cost- Efficient Task Scheduling Across Geo-Distributed Data Centers," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 3, pp. 705–718, 2018.

[14] "IBM ILOG CPLEX Optimizer," https://goo.gl/jyvDuV, accessed: 2017-10-16.

[15] "Capacity Scheduler," https://goo.gl/c9GS2p, accessed: 2017-10-16.

[16] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of Scientific Workflows," in *IEEE Workshop on Support of Large-Scale Science.*, 2008.

[17] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue MapReduce Benchmarks Suite," 2012. [Online]. Available: https://goo.gl/ccv2tK

[18] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-Constrained Workflow Scheduling Algorithms for Infrastructure as a Service Clouds," *Elsevier Future Generation Computer Systems*, vol. 29, no. 1, pp. 158–169, 2013.

[19] I. Cano, M. Weimer, D. Mahajan, C. Curino, and G. M. Fumarola, "Towards Geo-Distributed Machine Learning," *arXiv preprint arXiv:1603.09035*, 2016.

[20] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds," in *Proc. USENIX NSDI*, 2017.

[21] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling Jobs Across Geo-distributed Datacenters," in *Proc. ACM SoCC*, 2015.

[22] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing Data Parallel Jobs in Bandwidth-Skewed Environments," in *Proc. VLDB*, 2015.

[23] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: WAN-Aware Optimization for Analytics Queries," in *Proc. USENIX OSDI*, 2016.

[24] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a Geo-distributed Data-intensive World," in *Proc. CIDR*, 2015.

[25] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese, "Global Analytics in the Face of Bandwidth and Regulatory Constraints," in *Proc. USENIX NSDI*, 2015.

[26] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica, "Low Latency Geo-Distributed Data Analytics," in *Proc. ACM SIGCOMM*, 2015.