Leveraging Dependency in Scheduling and Preemption for High Throughput in Data-Parallel Clusters

Jinwei Liu*, Haiying Shen†, Ankur Sarker†, and Wingyan Chung‡
*Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, USA
†Department of Computer Science, University of Virginia, Charlottesville, VA, USA
‡Institute for Simulation and Training, University of Central Florida, Orlando, FL, USA
*jinweil@clemson.edu, †{hs6ms, as4mz}@virginia.edu, ‡wchung@ucf.edu

***Abstract*—Task scheduling and preemption are two important functions in data-parallel clusters. Though directed acyclic graph task dependencies are common in data-parallel clusters, previous task scheduling and preemption methods do not fully utilize such task dependency to increase throughput since they simply schedule precedent tasks prior to their dependent tasks or neglect the dependency. We notice that in both scheduling and preemption, choosing a task with more dependent tasks to run allows more tasks to be runnable next, which facilitates to select a task that can more increase throughput. Accordingly, in this paper, we propose a Dependency-aware Scheduling and Preemption system (DSP) to achieve high throughput. First, we build an integer linear programming model to minimize the makespan (i.e., the time when all jobs finish execution) with the consideration of task dependency and deadline, and derive the target server and start time for each task, which can minimize the makespan. Second, we utilize task dependency to determine tasks' priorities for preemption. Finally, we propose a method to reduce the number of unnecessary preemptions that cause more overhead than the throughput gain. Extensive experimental results based on a real cluster and Amazon EC2 cloud service show that DSP achieves much higher throughput compared to existing strategies.**

***Index Terms*—scheduling, task dependency, preemption, priority**

I. Introduction

An increasing number of large scale analytic frameworks [1] move towards high degree of parallelism to provide high throughput. For example, MapReduce, Cosmos and Spark are frameworks designed to process a large amount of data in parallel on a cluster of computing nodes. In such a data-parallel cluster, each job is partitioned to tasks and run on the cluster servers in parallel. Task scheduling and preemption are two important functions for high job performance.

Job scheduling is the process of assigning jobs to nodes (i.e., processors) in a manner to optimize the job performance. Usually, a user or a system submits jobs to the scheduler, which divides each job into tasks and forwards tasks to nodes for processing. A job usually consists of hundreds or thousands of concurrent tasks [2]. A job's completion time is determined by the completion time of the tail task. Placing a task on a contended node extends the completion time of the task. Therefore, the tasks of each job should be scheduled to appropriate nodes so that the job completion time can be reduced. Each node has a waiting queue which is used for queuing tasks when a node is allocated with more tasks than it can run concurrently [1, 3, 4]. In a preemption method, the priorities of tasks are determined, and each node chooses a high-priority task in its waiting queue to preempt its low-priority running task. The priorities are determined based on factors such as task's remaining time and waiting time to serve different performance objectives (e.g., high throughput, short job completion times).

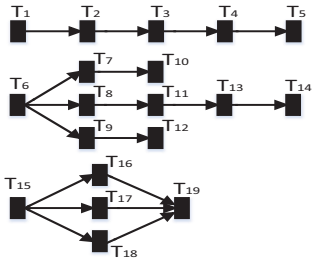


Fig. 1: Diverse dependency relations among tasks [6].

Dependency usually exists among tasks of a job. A task cannot start running until its precedent tasks complete. The big data problems such as machine learning and data mining in different areas (e.g., bioinformatics) involve complex computational dependencies [5]. Heterogeneous Directed Acyclic Graph (DAG) structured large and complex dependencies are increasingly common in data-parallel clusters, and the median DAG in such a cluster can have a depth of five and thousands of tasks [6]. Many previous methods have been proposed to detect the dependency between tasks [7–10, 4]. However, previous scheduling algorithms and preemption algorithms simply schedule precedent tasks prior to their dependent tasks or neglect the dependency.

The work in [7] indicates that task dependency needs to be considered in task scheduling for high throughput. Some schedulers [11–14, 6, 15–18] consider dependency. The sched-

2168-9253/18/$31.00 ©2018 IEEE
DOI 10.1109/CLUSTER.2018.00054

ulers first schedule runnable tasks, and leave the dependent tasks to the next scheduling. However, the runnable precedent tasks may complete a certain time period before the next scheduling, and then the server resources during this time period cannot be fully utilized. Also, if at next scheduling time t_1, there will be three idle slots, then assigning a task with three dependent tasks rather than a task with zero or four dependent tasks at scheduling time t_0 can more fully utilize resources. Only considering currently runnable tasks in scheduling without considering the dependent tasks in a global view may not be able to fully increase the throughput. Further, we need to consider the complex DAG structure. For example, in Figure 1, task T_6 has more dependent tasks than tasks T_1 and T_{15}. Executing T_6 at first can enable more dependent tasks to start executing after the precedent task T_6 completes, which helps increase the throughput. Though the works in [6, 19] consider the DAG dependencies, they do not handle the above issues. Therefore, we should judiciously determine tasks' execution order so that more tasks can start executing after a selected task completes and then the tasks that can more increase throughput can be selected to run.

Among the previous preemption methods [20–23], the works in [20, 21] choose the tasks that have the shortest remaining time to preempt, and the work in [22] further considers the waiting time to prevent task starvation, in which the task with the lowest priority may wait for a very long time (i.e., starvation). However, none of the works consider utilizing task dependency to increase the throughput. Also, when the priority is too fine-grained in the priority value space, there may be many preemptions. This leads to many context switchings and may reduce throughput. However, this issue was not handled in previous works.

In this paper, we explore how to fully utilize the DAG task dependency to increase throughput. It sheds light on another aspect that can be leveraged for performance improvement on data-parallel clusters. Specifically, we propose a Dependency-aware Scheduling and Preemption system (DSP), which is composed by an offline phase (scheduling) plus an online phase (preemption management). We summarize the contributions of this work below.

- DSP considers task dependency and assigns tasks to nodes so that independent tasks can run in parallel to minimize the makespan (i.e., the time when all jobs finish execution). We formulate an integer linear programming (ILP) problem to derive the target nodes and starting times for all tasks.
- DSP considers task dependency (along with remaining time, waiting time and deadline) to determine task priority for preemption. Specifically, DSP considers the number of dependent tasks in each level in DAG; more dependent tasks on higher levels lead to higher priority. Ultimately, this approach increases throughput by increasing the chance that more tasks can start executing after the selected task finishes execution.
- In DSP task preemption, a waiting task preempts a running task that has a lower priority than it, while avoiding unnecessary preemption that causes more overhead (due to context switch) than the throughput gain. In addition, all tasks are guaranteed to complete by their deadlines by preemption.

Extensive experiments on both real cluster and commercial cloud EC2 have been carried out to show DSP's advantages of improving throughput while satisfying the jobs' demands on completion time.

The remainder of this paper is organized as follows. Section II reviews the related work. Section III briefly introduces the background and concepts, and describes the throughput maximization model with the consideration of task dependency. Section IV-A illustrates how to determine task priority with the consideration of task dependency. Section IV-B describes the probabilistic based preemption. Section V presents the performance evaluation for DSP. Section VI concludes this paper with remarks on our future work.

II. Related Work

Many methods [2, 7, 24–27] have been proposed for job scheduling or task scheduling with the objective of maximizing throughput. However, all of these works for task or job scheduling neglect utilizing task dependency in scheduling to increase throughput. The work in [7] aims to maximize the throughput by fully utilizing the server resources when packing tasks to machines, and indicates that task dependency needs to be considered in task scheduling for high throughput.

Some works [6, 11–13, 19, 28] consider dependency in task or job scheduling. Grandl *et al.* [6] indicated that the long-running tasks and those with tough-to-pack resource demands should be focused on to decrease job completion time. Accordingly, they proposed GRAPHANE that consists of an offline and a a reconciliation heuristic online scheduler. Chowdhury and Stoic [11] proposed Aalo that strikes a balance and efficiently schedules coflows without prior knowledge. Ren *et al.* [12] designed Hopper, the first decentralized speculation-aware job scheduler that is provably optimal. Jalaparti *et al.* [13] developed Corral, a scheduling framework that uses characteristics of future workloads to determine an offline schedule. Su *et al.* [19] introduced a DAG-based non-preemptive task scheduling framework to dynamically map tasks to the most monetary cost efficient virtual machines based on the Pareto dominance solution (e.g., best possible task assignments) in order to reduce execution time and monetary cost. Unlike previous works, our work fully leverages task dependency to increase throughput in scheduling by prioritizing tasks that will subsequently enable the execution of more dependent tasks.

There is a large body of preemption work [14, 20–22] aiming at reducing the waiting time of high priority tasks. Chen *et al.* [14] developed preemption mechanisms to ensure low-latency and high resource utilization: an immediate preemption technique for short tasks without saving any states and a gradual preemption for long tasks to save their states. Ananthanarayanan *et al.* [20] proposed Amoeba which chooses

the tasks that consume the most resources to be preempted. Cho *et al.* [21] proposed Natjam which uses an on-demand checkpointing technique that saves the state of a task when it is preempted, so that it can resume where it left off when resources become available. However, all of these works suffer from the overhead of context switching, which decreases the throughput.

Unlike previous works that neglect the dependency in making preemption decision, DSP considers dependency and the combination of different factors (e.g., the number of dependent tasks, task's remaining time, task's waiting time) to determine task priority, which increases throughput and also avoids the long waiting time of low priority tasks. DSP further considers reducing the overhead of extensive number of preemptions for high throughput.

III. Dependency-aware Scheduling

In a distributed parallel computing system, there are nodes and schedulers. A job is split into m tasks, and the tasks are allocated to nodes. As in [6, 13, 3], our methods are applied to the scenario in which the jobs, task execution time and task dependencies of many tasks can be predicted a priori. Using DSP for the scheduling and preemption on this part of tasks still can improve the performance of the entire system. *Throughput* is the total number of jobs that complete their executions within their job deadlines during a unit of time.

We consider a scheduling problem with the objective of scheduling a group of jobs onto multiple nodes to maximize the throughput. Task dependency determines tasks' execution order. The dependency-aware scheduling is periodically executed offline after each unit of time period.

Suppose h jobs ($\mathcal{J} = \{J_1, ..., J_h\}$) are submitted in a unit of time period. Denote $\mathcal{L}_{\mathcal{MS}}$ as the makespan of these jobs. The deadlines on the completion time of the h jobs are represented by $t_1^d, ..., t_h^d$, respectively. We use T_{ij} to denote the jth task of J_i. Denote t_{ij}^s as the starting time of task T_{ij}, and t_{ij}^e as the completion time of task T_{ij}. Let $C_i^q = \{T_{ij}(1), ..., T_{ij}(|C_i^q|)\}$ ($C_i^q \in \mathbf{C}_i, q \in \{1, ..., |\mathbf{C}_i|\}$) be an arbitrary chain of tasks belonging to job J_i, representing the dependency of the tasks, where $|C_i^q|$ represents the length of the chain C_i^q, and $\mathbf{C}_i$ is the set of chains of tasks belonging to job J_i. All tasks $T_{ij} \in C_i^q$ ($i \in \{1, ..., h\}, j \in \{1, ..., m\}$) must be processed sequentially one after another. Denote $T_{ij}(k)$ as the kth task on the chain C_i^q belonging to job J_i, and $T_{ij}(k-1)$ as the $(k-1)$th task (precedent task of $T_{ij}(k)$) on the chain C_i^q. Task $T_{ij}(k)$ cannot start executing until task $T_{ij}(k-1)$ finishes execution, where $k \in \{1, ..., |C_i^q|\}$. Denote $t_{ij}^s(k)$ and $t_{ij}^e(k)$ as the starting time and completion time of the kth task on the chain C_i^q belonging to job J_i running on a node. For easy reference, Table I shows the main notations used in this paper.

Denote $x_{ij,k} = \{0, 1\}$ as an indicator variable; $x_{ij,k} = 1$ if task T_{ij} is assigned to node k, otherwise $x_{ij,k} = 0$. Define $g(k)$ ($k \in \{1, ..., n\}$) as a function of processing rate of the kth node, which is the million instructions per second (MIPS) speed, and $g(k)$ is related to the CPU size s_{cpu}^k and memory size s_{mem}^k of the node k. Specifically, $g(k)$ is expressed as follows

TABLE I: Notations.

$\mathcal{J}$	A set of jobs	C_i^q	A chain of tasks belonging to J_i
h	# of jobs in $\mathcal{J}$	$\|C_i^q\|$	The length of chain C_i^q
J_i	The ith job in $\mathcal{J}$	$g(k)$	Proc. rate func. of node k
t_i^d	Job J_i's deadline	l_{ij}	Task T_{ij}'s size
T_{ij}	The jth task of J_i	T_i	Task i
t_{ij}^s	T_{ij}'s starting time	$\mathcal{M}$	A set of target nodes
t_{ij}^e	T_{ij}'s ending time	$t_{ij,k}$	T_{ij}'s Exec. time on node k
t_{ij}^r	T_{ij}'s rec. time / preemption	n	Total # of nodes
s_{mem}^k	Mem. size of node k	m	# of tasks of a job
s_{cpu}^k	CPU size of node k	N_{ij}^p	# of preemptions for T_{ij}
t_{ij}^{rem}	T_{ij}'s remaining time	P_{ij}^t	T_{ij}'s priority at time t

$$g(k) = \theta_1 s_{cpu}^k + \theta_2 s_{mem}^k \tag{1}$$

where θ_1 and θ_2 are the weights for CPU size and memory size, respectively. Denote $t_{ij,k}$ as the time period for executing task T_{ij} on node k without being preempted. Thus, we have

$$t_{ij,k} = l_{ij}/g(k) \tag{2}$$

where l_{ij} is the size of task T_{ij} in terms of Millions of Instructions (MI). Suspending a running task and putting a waiting task to running state consumes a certain amount of time (called recovery time) due to the context switching. Let t_{ij}^r be the recovery time of a preemption for task T_{ij}, and $N_{ij,k}^p$ be the number of preemptions for task T_{ij} at node k. $N_{ij,k}^p$ of a task can be estimated based on its size, dependency, and deadline using the method introduced in [29]. Denote σ as the threshold for the time (i.e., 0.05s) that an *evicted* task should wait for starting its execution after it has been selected for the execution. Let $y_{ij,uv,k} = 1$ if task T_{ij} precedes task T_{uv} on node k, otherwise $y_{ij,uv,k} = 0$. Since linear programming (LP) is considered more efficient in solving scheduling problems [30], we formalize our problem as the following ILP problem:

$$Min\{\mathcal{L}_{\mathcal{MS}}\} \tag{3}$$

$$s.t.\ \max\left(t_{ij}^s + t_{ij,k} \cdot x_{ij,k} + N_{ij,k}^p (t_{ij}^r + \sigma) x_{ij,k}\right) - \min t_{ij}^s \leq \mathcal{L}_{\mathcal{MS}} \quad (\forall i \in \{1, ..., h\}, j \in \{1, ..., m\}, k \in \{1, ..., n\}) \tag{4}$$

$$(t_{ij}^s + t_{ij,k}) \cdot x_{ij,k} \leq (t_{uv}^s + (1 - y_{ij,uv,k}) \cdot t_{uv,k}) \cdot x_{uv,k} \quad (\forall i, u \in \{1, ..., h\}, j, v \in \{1, ..., m\}, k \in \{1, ..., n\}) \tag{5}$$

$$\max\left(t_{ij}^s + t_{ij,k} \cdot x_{ij,k} + N_{ij,k}^p (t_{ij}^r + \sigma) x_{ij,k}\right) \leq t_i^d \quad (\forall j \in \{1, ..., m\}, k \in \{1, ..., n\}, T_{ij} \in C_i^q, C_i^q \in \mathbf{C}_i) \tag{6}$$

$$\max\left(t_{ij}^s + t_{ij,l} \cdot x_{ij,l} + N_{ij,l}^p \cdot (t_{ij}^r + \sigma) \cdot x_{ij,l}\right) \leq t_{iq}^s \quad (k \in \{1, ..., m\}, l \in \{1, ..., n\}, T_{ij} \in C_i^q, C_i^q \in \mathbf{C}_i) \tag{7}$$

$$y_{ij,uv,k} + y_{uv,ij,k} = 1 \ (x_{ij,k} = 1, x_{uv,k} = 1) \tag{8}$$

$$y_{ij,uv,k} \in \{0, 1\} \tag{9}$$

$$x_{ij,k} \in \{0, 1\} \tag{10}$$

$$t_{ij}^s \geq 0 \tag{11}$$

Constraint (4) is to ensure the time consumed by the job completing at last is no longer than the makespan. Constraint (5) is to ensure the execution order between T_{ij} and T_{uv} on node k when T_{ij} is already running and T_{uv} is a newly assigned task on node k. Constraint (6) is to ensure jobs can finish execution within their specified deadlines, where $\mathbf{C}_i$ is the set of chains of tasks belonging to job J_i. Constraint (7) is to ensure the dependency relation between T_{ij} and T_{iq}, that is, task T_{ij} precedes task T_{iq} on node k. In addition, we adopt a checkpoint-restart mechanism [29], in which preempted tasks are restarted from their most recent checkpoints.

The target node $k|_{x_{ij,k}=1}$ ($k|_{x_{ij,k}=1}$ represents the node who will be assigned the task T_{ij}, where $x_{ij,k} = 1$ if T_{ij} is assigned to node k) for task T_{ij} must be an integer in practice. To make our formulated optimization problem more tractable, we can first relax the problem to a real-number optimization problem in which $k|_{x_{ij,k}=1}$ can be a real number, and derive the solution for the real-number optimization problem. Then, we can use integer rounding to get the solution for practical use. Considering the task scheduling problem in general is NP-complete [10], we use the CPLEX linear program solver [31] to solve this ILP problem.

Given a set of jobs consisting of tasks, constraints of jobs and tasks (i.e., task dependency constraints, job completion time constraints), and a number of nodes, the output of the problem solution provides the task schedule [t^s_{ij}, $k|_{x_{ij,k}=1}$] ($\forall i \in \{1, ..., h\}, j \in \{1, ..., m\}, k \in \{1, ..., n\}$), that is, the target node (denoted by $k|_{x_{ij,k}=1}$) and the starting time (denoted by t^s_{ij}) for each task. Then, the minimized makespan (denoted by $\mathcal{L}_{\mathcal{MS}}$) can be obtained from the ILP model. Based on the output of the ILP model [t^s_{ij}, $k|_{x_{ij,k}=1}$], the schedulers assign tasks to their target nodes by following the order of their starting times.

Dependency-aware scheduling in DSP is an offline approach. It runs periodically for all the jobs submitted during a unit period of time. Since the jobs, task dependency, task execution time, task completion time and the number of preemptions of a task are estimated, and also a node sometimes may not provide sufficient resources for task running, the actual dependency and task completion time may not be the same as the estimated. We propose an adaptive online scheduling procedure, i.e., dependency-aware preemption, to adjust the schedule dynamically based on the evaluation of task priority. Specifically, we partition a unit period of time to several smaller time periods (called epochs). In each epoch, DSP evaluates the task priority and conducts preemptions. Note that the dependency-aware scheduling has a relatively high time overhead, but since it is offline, it will not affect the system job running performance. However, if the high time overhead of the offline method is a concern for a data-parallel cluster, then it can only run the online dependency-aware preemption method to achieve high throughput.

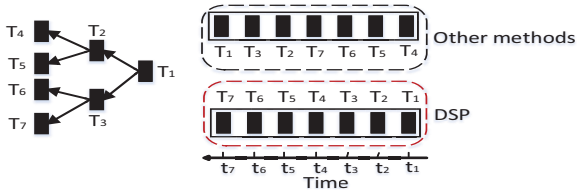


Fig. 2: Determining task priority by considering task dependency.

IV. Dependency-aware Task Preemption

A. Dependency-aware Task Priority Determination

It has been shown that assigning higher priority to tasks with shorter remaining time increases the throughput [32]. However, this also increases the waiting time of large-size tasks. To avoid starvation of large-size tasks and reduce the waiting time for each task, previous preemption methods define task priority based on task remaining time and (or) waiting time. However, they neglect dependency in the priority determination in preemption, which is an important factor to consider in preemption to maximize the throughput. Choosing a task with more dependent tasks to run enables more tasks to be runnable next, and more runnable task options enable to select a better task that can more increase the throughput. In this section, we propose task priority that considers task dependency to improve throughput.

Figure 2 shows an example illustrating the importance of considering dependency in preemption. There are 7 tasks $T_1, ..., T_7$. T_2-T_3, T_4-T_5 and T_6-T_7 depend on T_1, T_2 and T_3, respectively. Without considering task dependency, other methods may assign priorities to tasks by following: T_1<T_3<T_2<T_7<T_6<T_5<T_4. In this case, T_1 has the lowest chance to be executed. But all the other tasks cannot start execution if T_1 does not finish execution, and even worse, deadlock may occur due to the dependency constraints, which can increase tasks' waiting time and decrease the throughput. By considering dependency, task remaining time and waiting time in priority determination, DSP assigns priorities to tasks by following: T_7<T_6<T_5<T_4<T_3<T_2<T_1 or T_6<T_7<T_5<T_4<T_3<T_2<T_1, etc. Thus, DSP helps increase throughput.

Below, we first show an example to indicate the importance of considering different aspects in dependency DAG in preemption, and then propose a method to determine task priority in order to increase the throughput while reducing the average waiting time for each task. Assigning higher priority to the task that has more dependent tasks can increase the chance that more tasks depending on it can start executing after the task finishes execution. As shown in Figure 3, there are different kinds of dependencies [24]. Although T_1 and T_6 have the same number of dependent tasks, T_6 has higher priority than T_1 for execution because more tasks can be executed soon after T_6 finishes execution and a better option can be chosen from the candidates. Similarly, T_{11} should have a higher priority than T_6. Task T_{11} has more dependent tasks compared with the other tasks (e.g., T_1, T_6). Although T_{11} and T_6 have the same

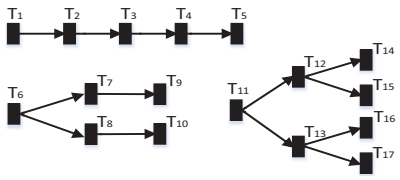


Fig. 3: Utilize task dependency to determine task priority.

number of dependent tasks in the first level which is more than T_1, T_{11} has more dependent tasks in the second level than T_6. In this case, executing T_{11} at first helps increase the throughput. Thus T_{11} has higher priority than other tasks considering dependency.

Based on the queuing theory, choosing a task with shortest remaining time to preempt the running task can increase the throughput [32]. Therefore, we also consider assigning higher priority to a task with less remaining time. Since shortest remaining first can easily incur the starvation of tasks with longer remaining time, we also consider task waiting time to determine task priority to avoid starvation in order to increase the throughput.

Denote t_{ij}^{rem} as the remaining time of task T_{ij}, and t_{ij}^{w} as the waiting time of task T_{ij}. To determine the priority of a task T_{ij}, we consider it as a function of its remaining time t_{ij}^{rem} and its waiting time t_{ij}^{w} based on the dependency relations among T_{ij} and its children (i.e., the tasks depending on T_{ij}) considering the above indicated aspects.

Given a particular and arbitrary task T_{ij}, the priority of task T_{ij} at time t is recursively computed as follows:

$$P_{ij}^{t} = \sum_{T_{ik} \in S_{ij}} (\gamma + 1) P_{ik}^{t} \tag{12}$$

where S_{ij} represents a set consisting of T_{ij}'s children, and $\gamma \in (0, 1)$ is a coefficient that is used to give higher priority to higher levels. For a given task (e.g., T_{ij}) that has no dependent tasks, its priority at time t is computed in below:

$$P_{ij}^{t} = \omega_1 \cdot \frac{1}{t_{ij}^{rem}} + \omega_2 \cdot t_{ij}^{w} + \omega_3 \cdot t_{ij}^{a} \tag{13}$$

where t_{ij}^{a} is the allowable waiting time of task T_{ij}, ω_1, ω_2 and ω_3 are the weights for the task's remaining time, waiting time and allowable waiting time, respectively, and $\omega_1 + \omega_2 + \omega_3 = 1$. The priority of task T_{ij} at time t_1 ($P_{ij}^{t_1}$) can be obtained based on Formulas (12) and (13). Our task priority determination method utilizes task dependency to recursively calculate a task's priority based on its children's priorities.

B. Task Preemption Procedure

Recall that the offline dependency-aware scheduling of DSP in Section III outputs the target node and the starting time for each task. As a result, as shown in Figure 4, each node has a queue containing the tasks that will run on this node. In the figure, the tasks with the same color belong to the same job. The tasks in a queue are in the ascending order of their starting times; a task with an earlier starting time will run earlier. The online task preemption is executed after each epoch in a unit of time period. Recall that the online preemption is used to adjust the schedule dynamically due to insufficiently accurate estimation of some parameters in scheduling and dynamically changing job running environment. Since the output is approximately close to the optimal solution, we only need to consider the first a few waiting tasks (say δ percent of all tasks) (called preempting tasks) rather than all tasks in a queue in order to save overhead. The value of δ can be dynamically adjusted based on the percent of the considered waiting tasks that preempt the running tasks. A larger percent means that more adjustments need to be made on the offline schedule and we then can increase δ, and vice versa. We need to ensure that each task completes by its deadline. Therefore, among the waiting tasks in a queue, we also find the tasks that have allowable waiting time before their deadlines no larger than ϵ (a very small number) to preempt running tasks. We call these waiting tasks *urgent tasks* that need to run immediately.

Next, we introduce how to calculate a task's allowable waiting time. Recall that we aim to complete each job J_i by its deadline. Then, each of its task T_{ij} has a deadline. Only when all of its tasks meet their deadlines, the job can meet its deadline. The dependency DAG of a job has different levels as shown in Figure 3 and we use L to denote the total number of levels in the DAG. We use t_{ij} to denote the execution time of task T_{ij}, use t_{ijk} to denote the execution time of task j at the k^{th} level, and use t_{ij}^{d} to denote the deadline of task j of job J_i. The deadline of the tasks in the last level is the job's deadline, i.e., $t_{ijL}^{d} = t_i^{d}$. Then, the deadline of the tasks in the l^{th} level is the job's deadline subtracted by the maximum execution time of the tasks in each level from the last level to the $(l+1)^{th}$ level, i.e., $t_{ij}^{d} = t_i^{d} - \sum_{k=l+1}^{L} \max_j\{t_{ijk}\}$. Then, a task's allowable waiting time (denoted by t_{ij}^{a}) is $t_{ij}^{a} = t_{ij}^{d} - t_{ij}^{rem}$. This means as long as the task's subsequent waiting time is no greater than t_{ij}^{a}, it can complete by its deadline.

Below, we introduce how to conduct preemption between the urgent tasks and preempting tasks, and the running tasks. Among the running tasks, those tasks that have allowable waiting time larger than epoch time are preemptable running tasks. This is to make sure that tasks will not miss their deadlines caused by being preempted. The preempted tasks are inserted to the queue based on their starting time that keeps all queueing tasks in the ascending order of starting time. In the preemption procedure, for each queue, we first find the urgent tasks. Each urgent task preempts the running task with the lowest priority (that the urgent task does not depend on) among the preemptable running tasks in sequence. Among the δ percent of all tasks at the beginning of the queue (i.e., preempting tasks), for each task, we check if it can preempt a running task based on the priority.

We sort the preemptable running tasks in the ascending order of priority. We pick the first running task from the sorted list T_j. Then, waiting task T_i can preempt running task T_j if the following conditions are satisfied.

• Condition C_1: The priority of the waiting task is higher than that of the running task.

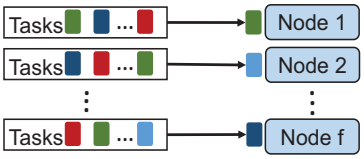


Fig. 4: Preemption for multiple tasks running on multiple processors.

• Condition C_2: The waiting task does not depend on the running task.

If these two conditions are not satisfied, the algorithm chooses the second lowest priority running task and determines if the running task can be preempted by waiting task T_i, and so on. However, if the waiting time of a task is longer than the threshold τ (e.g., 0.05s), it preempts the running task no matter conditions C_1 is satisfied or not. This process repeats until T_i preempts a running task, Condition C_1 is not satisfied (i.e., the remaining preemptable running tasks cannot be preempted by T_i) or all preemptable running tasks are checked. Then, the algorithm chooses the next preempting task and repeats the above process to preempt a running task. When all preempting tasks are checked or all preemptable running tasks are preempted, this process stops.

In the above algorithm, as in previous preemption methods, it simply compares the priorities of a waiting task and a running task, and the preemption occurs if the waiting task's priority is higher than the running task. Consider a case that preemptions occur frequently but the priority of the preempting task is only slightly higher than that of the preempted task. This generates much time overhead for context switching and may increase job completion time and decrease the throughput. We use an example to illustrate this case. Suppose the priority of the current running task T_1 is 1.1, when a task (T_2) with priority 1.7 is considered in the waiting queue, then T_2 preempts T_1. Soon after T_2 starts running, another task T_3 with priority 2 is considered in the waiting queue, then T_3 preempts T_2. Such frequent preemptions lead to many context switches, which extends job completion time and possibly reduces throughput. Since T_2 and T_3 have close priority, the throughput loss caused by preemption may offset or may be even higher than the throughput increase caused by running T_3 first. In this case, the preemption is not necessary. Next, we propose a normalized priority method to avoid such unnecessary preemption that causes more overhead (due to context switch) than the throughput gain. Instead of checking the absolute value of the priority difference between a preempting task and a preemptable task (denoted by $\hat{P}$), we check its normalized value (denoted by $\tilde{P}$) by the average difference between neighboring tasks among all tasks sorted based on priority. First, we order all the tasks in ascending order of their priorities. Then, we calculate the priority difference between each pair of neighboring tasks and finally calculate the average difference (denoted by $\bar{P}$). The normalized value is calculated by $\tilde{P} = \hat{P}/\bar{P}$, which represents the scale of the priority difference globally. We specify that only when $\tilde{P} > \rho\hat{P}/\bar{P}$ ($\rho > 1$), the preempting task can preempt the preemptable task, where the $\rho\hat{P}$ increase of the priority of a running task leads to throughput increase comparable to the overhead of the context switch of the preemption. The value of ρ is set empirically. We ensure that the priority difference between a preempting task and a preemptable task must be larger than the global average difference for premption. This way, each preemption brings about considerably more throughput than the overhead cause by the context switch. Algorithm 1 shows the pseudocode for the dependency-considered task preemption algorithm.

Algorithm 1: Pseudocode for DSP task preemption

```
Input: A is the set of waiting tasks and B is the set of running tasks
Compute the priorities of all waiting and running tasks
Sort B in ascending order based on their priority values
for each i = 1, ..., |A| do
    if t^a[i] ≤ ε OR t^w[i] ≥ τ then
        for each j = 1, ..., |B| do
            if A[i] does not depend on B[j] then
                Suspend task B[j] and run task A[i]
                break
            else
                j ← j + 1
                continue
for each i = 1, ..., |δ|A|| do
    for each j = 1, ..., |B| do
        if A[i] depends on B[j] then
            j ← j + 1
            continue
        Compute priority difference P̂_i and normalized priority P̃_i of A[i]
        if P̂_i > 0 && P̃_i > ρP̂/P̄ then
            Suspend task B[j] and run task A[i]
            break
```

Complexity of DSP: DSP system consists of two parts: the ILP model for deriving the target nodes for tasks and the priority based preemption (PP) for preemption. Based on [33], systems of linear inequalities with at most two variables per inequality can be decided in strongly polynomial time. For PP, the time complexity is $O(kf)$.

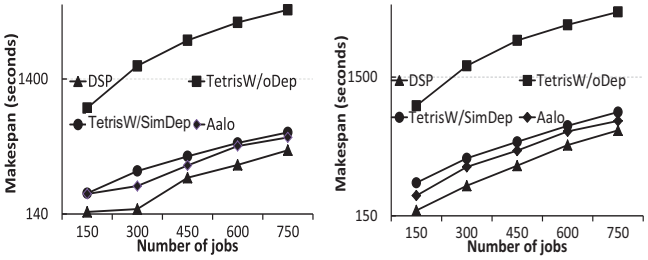
V. Performance Evaluation

In this section, we introduce our experimental results on a large-scale real cluster Palmetto [34], which is Clemson University's primary high-performance computing (HPC) resource, and Amazon EC2 [35], respectively.

We use *TetrisW/SimDep* to denote the Simple Dependency-aware scheduling method introduced in Tetris [7], in which precedent tasks complete before their dependent tasks start to run. To test DSP's performance on dependency-aware scheduling, we compared DSP with *TetrisW/SimDep*, Tetris [7] without any dependency consideration (denoted by *TetrisW/oDep*) and Aalo [11]. To test DSP's performance on preemption, we compared DSP with three other methods, Amoeba [20],

TABLE II: Parameter settings.

Parameter	Meaning	Setting
n	# of servers	30-50
h	# of jobs	150-2500
m	# of tasks of a job	100-2000
δ	Minimum required ratio	0.35
τ	Threshold of tasks' waiting time for execution	0.05
θ_1	Weight for CPU size	0.5
θ_2	Weight for Mem size	0.5
α	Weight for waiting time for SRPT	0.5
β	Weight for remaining time for SRPT	1
γ	A certain coefficient $\in (0, 1)$	0.5
ω_1	Weight for task's remaining time	0.5
ω_2	Weight for task's waiting time	0.3
ω_3	Weight for task's allowable waiting time	0.2



(a) On the real cluster (b) On Amazon EC2

Fig. 5: Performance of various methods on makespan vs. the number of jobs.

Natjam [21] and SRPT [22]. We choose these three methods because like DSP, they also aim at increasing the throughput.

Tetris [7]. Tetris aims to maximize task throughput and speed up job completion by packing tasks to machines based on their resource requirements [36]. Specifically, when resources on a machine become available, it first selects the set of tasks whose peak usage of each resource can be accommodated on that machine. It then computes an alignment score (i.e., a weighted dot product between the vector of the machine's available resources and the task's peak usage of resources) on which the set of tasks whose peak usage of each resource can be accommodated. The task with the highest alignment score (i.e., priority) is scheduled to the machine and allocated with its peak resource demands.

Aalo [11]. Aalo aims to minimize the average coflow's completion time without prior knowledge of coflow characteristics. Coflow means a collection of network data flows that share a common performance goal, e.g., minimizing the completion time of coflow. Aalo puts all flows of a coflow in the same queue to satisfy the dependency constraint, and all coflows are distributed to several queues. The flows in each queue are sent out in the First-In-First-Out (FIFO) order. The global coordination for the scheduling order among different queues is conducted using the method in [37] to achieve fairness among coflows. Aalo does not consider the deadlines of coflows. In our implementation, we consider a job as a coflow and the task as the flows in the coflow.

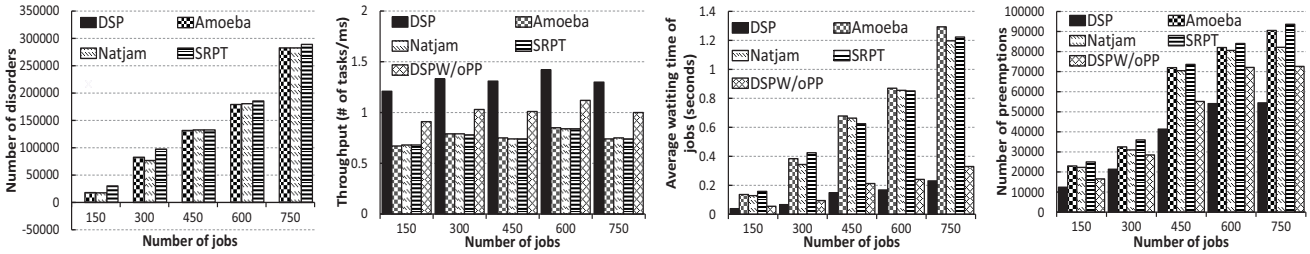
Amoeba [20]. The task that needs the most resources (i.e., longest remaining time [21]) has the lowest priority and vice versa in preemption to increase the overall throughput. Amoeba uses a checkpointing mechanism in task preemption, in which tasks are restarted from their most recent checkpoints.

Natjam [21]. Natjam assigns higher priority to production jobs and lower priority to research jobs in scheduling. It uses production jobs to preempt research jobs. For an arrival production job, Natjam selects a research job for eviction that uses the most resources firstly, that has the maximum deadline secondly, and that has the shortest remaining time thirdly. Also, it uses a checkpointing mechanism.

SRPT [22]. SRPT prioritizes jobs and we used it for prioritizing tasks. It uses the linear combination of waiting time and the remaining time for a task (i.e., estimated time for completing the remaining part of the task) to determine the priority of a task. SRPT does not use a checkpoint mechanism. As in [22], we set the weight of waiting time α to 0.5 and the weight of remaining time β to 1 to calculate job priority.

We first deployed our testbed on 50 servers in a large-scale real cluster located in our university. The servers in the real cluster are from Sun X2200 servers (AMD Opteron 2356 CPU, 16GB memory). We then conducted experiments on 30 instances in the real-world Amazon EC2 and the instances in EC2 are from commercial product HP ProLiant ML110 G5 servers (2660 MIPS CPU, 4GB memory). We considered each instances as a server. Each server (instance) was set to have 1GB/s bandwidth and 720GB disk storage capacity in both real cluster and EC2 experiments.

In each experiment, we varied the number of jobs from 150 to 750 with step size of 150. The job arrival rate was set to x jobs per minute and x was randomly chosen from [2,5] [38]. We defined three categories of jobs based on the number of tasks of a job. A large job has 2000 tasks, a medium job has 1000 tasks and a small job has several hundreds of tasks [7]. All jobs in an experiment consist of the equal number of small, medium, and large jobs. The Google Cluster trace [38] records resource usage on a cluster of about 11000 machines from May 2011 for 29 days. We randomly chose tasks from the jobs in the period between May 1 to May 7. The CPU and memory consumption, and execution time for each task were set based on the Google cluster trace, and the disk and bandwidth consumption for each task was set to 0.02MB [39] and 0.02 MB/s [40, 41], respectively. In the experiment, we created the dependency relationship among tasks based on their starting time and ending time from the trace. When there is no overlap between the execution times of two tasks of a job, we can create a dependency relationship between the two tasks. We constrained the number of levels in a created dependency DAG within five and the number of dependent tasks on a task within fifteen [6]. In the experiment, we ran the scheduling periodically every 5mins. If the resources of a server were not enough to run waiting tasks, we conducted preemption. Table II shows the parameter settings in our experiment unless otherwise specified.



(a) The number of disorders (b) Throughput (c) Jobs' average waiting time (d) Overhead

Fig. 6: Performance of different evaluation metrics versus the number of jobs of different methods on the real cluster.

A. Performance Comparison with Scheduling Methods

Figure 5(a) and Figure 5(b) show the relationship between the makespan and the number of jobs on the real cluster and Amazon EC2, respectively. In both figures, we see that the makespan increases as the number of jobs increases. This is because with a fixed number of nodes, the more jobs submitted, the more time the nodes need to finish executing all jobs. Moreover, the makespans follow DSP<Aalo<TetrisW/SimDep<TetrisW/oDep.

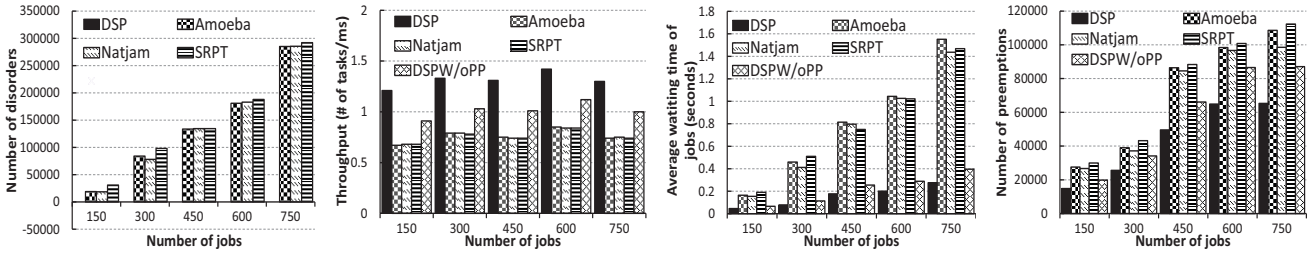
TetrisW/oDep does not consider dependency in scheduling. TetrisW/SimDep simply schedules precedent tasks before scheduling their dependent tasks. Aalo considers dependencies between tasks, but it does not consider deadline constraints of tasks. DSP uses an ILP model to minimize the makespan by fully utilizing task dependency information. The ILP problem solution assigns tasks that are independent of each other to different nodes so that the tasks can run in parallel, and thus increases the throughput and decreases the makespan. The solution gives priority to the tasks that have more dependent tasks in scheduling, which makes more tasks runnable next in scheduling. As a result, DSP generates the least makespan and high throughput.

B. Performance Comparison with Preemption Methods

This section measures the performance of the preemption methods. We use our initial schedule for all preemption methods. Figure 6(a) shows the relationship between the number of disorders (execution order is inconsistent with task dependency relation) and the number of jobs. We see that the number of disorders for DSP is always 0 and the result follows DSP<Natjam≈Amoeba<SRPT. The reason is that DSP considers the dependencies among tasks when it preempts tasks, however the other methods do not consider the dependencies among tasks when they schedule tasks.

To show the advantages of normalized priority based preemption (PP), we evaluated the performance of DSPW/oPP, a variant of DSP in which the PP is not used. DSPW/oPP conducts preemption based on absolute value of task priority like the compared methods. Figure 6(b) shows the relationship between the throughput measured by tasks/ms and the number of jobs. We see the throughput follows SRPT<Amoeba≈Natjam<DSPW/oPP<DSP. This is because (1) DSP utilizes task dependency information to determine task priority, and assigns higher priority to tasks that have more dependent tasks in higher levels, so running higher priority tasks enables more runnable tasks to run sooner, which helps increase the throughput. (2) DSP considers deadline constraints for job completion time. (3) DSP considers the dependencies among tasks in preemption. (4) DSP uses PP to reduce the time overhead caused by preemption and thereby increase the throughput. Without using PP, DSPW/oPP produces less throughput than DSP. However, all three other methods schedule the tasks based on their priorities and do not consider dependency in preemption. Then, some tasks that depend on other tasks need to wait for other tasks to finish, so the methods produce lower throughput than DSP. Throughput is affected by the remaining time. Amoeba only considers the remaining time to choose tasks, so it has relatively higher throughput. Natjam also considers task remaining time to choose task for eviction. Moreover, it takes into account job deadlines for scheduling tasks. Thus it has relatively higher throughput. In addition to considering remaining time like other methods to increase throughput, SRPT also considers task waiting time to avoid task starvation. Since SRPT does not use the checkpoint mechanism, a preempted task must be restarted from the scratch rather than from its most recent checkpoint, thus reducing throughput.

Figure 6(c) reveals the relationship between the average waiting time of jobs and the number of jobs. We see the average waiting time of jobs approximately follows DSP<DSPW/oPP<Natjam≈SRPT<Amoeba. This is because DSP utilizes the task dependency information to determine task priority for execution. Also, DSP considers task waiting time for determining task priority, which reduces the waiting time of low priority tasks and also can avoid starvation. In addition, DSP uses a probabilistic based approach for preemption (ignored in DSPW/oPP), which can reduce the time overhead caused by preemption and thus reduce the waiting time of tasks. Therefore DSP has the shortest average waiting time, followed by DSPW/oPP. The average waiting times of jobs of Natjam and SRPT are nearly the same, but relatively shorter than that of Amoeba. This is because Natjam considers job deadlines and SRPT considers task waiting time to determine task priority for preemption, which can reduce



(a) The number of disorders (b) Throughput (c) Jobs' average waiting time (d) Overhead

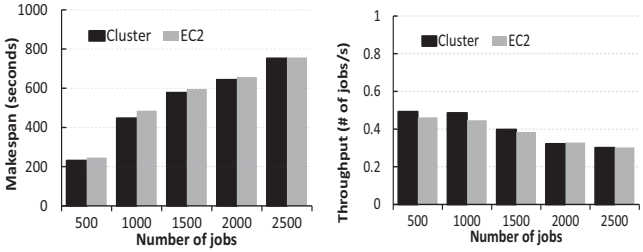
Fig. 7: Performance of different evaluation metrics versus the number of jobs of different methods on Amazon EC2.

the average waiting time of jobs. However, Amoeba neither considers task waiting time for determining task priority nor has deadline constraints for jobs, thus it has relative longer average waiting time of jobs.

Figure 6(d) shows the relationship between the number of preemptions and the number of jobs for scheduling. We can measure the execution time overhead due to context switching based on the number of preemptions as in [42]. From the figure, we observe that the number of preemptions follows DSP<DSPW/oPP<Natjam<Amoeba<SRPT. This is because (1) In DSP, a running task could be preempted only if the waiting task does not depend on the running task. (2) DSP allows preemptions only for a part of waiting tasks while Natjam, Amoeba and SRPT allow preemptions for all tasks in the waiting queue, which increases the number of preemptions. As a result, Natjam, Amoeba and SRPT have relatively more preemptions than DSP and DSPW/oPP. Since PP reduces the number of unnecessary preemptions that generate overhead higher than the throughput gain, DSP produces fewer preemptions than DSPW/oPP. Also, Natjam supports preemption for only research jobs rather than production jobs. Thus it generates fewer preemptions than Amoeba and SRPT. Unlike all the other methods, SRPT does not use a checkpoint mechanism, so that preempted tasks must start from the beginning and experience preemption more often, which increases the number of preemptions of these tasks.

To further verify the performance of DSP, we also conducted experiments on Amazon EC2. The experimental results are shown in Figure 7(a), 7(b), 7(c) and 7(d), which mirror Figure 6(a), 6(b), 6(c) and 6(d), respectively, due to the same reasons explained previously.

Comparing Figure 7(c) with Figure 6(c), we find the average waiting time of jobs in Figure 7(c) is longer than that in Figure 6(c). The reason is that the number of nodes in the real cluster environment is larger than that in Amazon EC2, and tasks have more chances to find idle nodes. Comparing Figure 7(d) to Figure 6(d), we find the number of preemptions in Figure 7(d) is relatively higher than that in Figure 6(d). This is because the average number of tasks assigned to a node in the real cluster environment is less than that of Amazon EC2 due to less nodes in Amazon EC2, and preemption is more likely to occur for nodes with more tasks.



(a) Makespan (b) Throughput

Fig. 8: Performance on scalability of DSP.

C. Evaluation of Scalability

To test the scalability of our DSP system, we varied the number of jobs from 500 to 2500 with step size 500, and measured its makespan and throughput in the real cluster and Amazon EC2. Figure 8(a) shows the makespan and Figure 8(b) shows the throughput on the real cluster and Amazon EC2. we see that the makespan gradually increases and the throughput gradually decreases as the number of jobs increases. Both do not change dramatically when the number of jobs becomes very large. Therefore, the DSP system is scalable.

VI. Conclusion

DAG structured large and complex task dependencies are increasingly common in data-parallel clusters. However, previous scheduling methods and preemption methods do not fully leverage the complex dependencies to improve system throughput; previous scheduling methods simply schedule the precedent tasks before their dependent tasks, while previous preemption methods neglect dependency. In this paper, we propose Dependency-aware Scheduling and Preemption system (DSP), which is the first work that fully leverages the dependency in scheduling and preemption to improve throughput. DSP prioritizes tasks that will subsequently enable the execution of more dependent tasks and hence the selection of tasks that more increase throughput. DSP judiciously utilizes task dependency information and jointly considers task remaining time and waiting time for determining task priority.

In preemption, DSP preempts lower priority tasks with higher priority tasks and also ensures that tasks complete by their deadlines. It further has a normalized priority method to avoid unnecessary preemption that causes more overhead (due to context switch) than the throughput gain. We compare DSP with other existing methods using a real cluster and Amazon EC2 cloud service, and demonstrate that DSP outperforms the existing methods in terms of throughput, preemption overhead, job waiting time and following task dependency relationship. In our future work, we will consider data locality, fairness, cross-job dependency, and the scenario that new tasks are dynamically added which extends the task-dependency graph. We will also study the sensitivity of the parameters. Further, we will consider fault tolerance in designing a dependency-aware scheduling and preemption system [43] so that the system can handle node failures/crashes or straggler.

ACKNOWLEDGMENT

This research was supported in part by U.S. NSF grants OAC-1724845, ACI-1719397 and CNS-1733596, and Microsoft Research Faculty Fellowship 8300751.

REFERENCES

[1] E. Boutin, J. Ekanayake, W. Lin, B. Shi, and J. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proc. of OSDI*, 2014.

[2] K. Ousterhout, M. Wendell, P.and Zaharia, and I. Stoica. Batch sampling: Low overhead scheduling for sub-second parallel jobs. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.294.2048, 2012. [accessed in July 2017].

[3] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proc. of ATC*, 2015.

[4] J. Liu and H. Shen. Dependency-aware and resource-efficient scheduling for heterogeneous jobs in clouds. In *Proc. of CloudCom*, 2016.

[5] H. Kashyap, H. Ahmed, N. Hoque, S. Roy, and D. Bhattacharyya. Big data analytics in bioinformatics: A machine learning perspective. *arXiv preprint arXiv:1506.05101*, 2015.

[6] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. of OSDI*, 2016.

[7] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proc. of SIGCOMM*, 2014.

[8] V. Vavilapalli, A. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop YARN: Yet another resource negotiator. In *Proc. of SoCC*, 2013.

[9] L. Zheng, C. Zeng, L. Li, Y. Jiang, W. Xue, J. Li, C. Shen, W. Zhou, H. Li, L. Tang, T. Li, B. Duan, M. Lei, and P. Wang. Applying data mining techniques to address critical process optimization needs in advanced manufacturing. In *Proc. of KDD*, 2014.

[10] H. Topcuoglu and S. Hariri and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. on TPDS*, 13(3), 2002.

[11] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *Proc. of SIGCOMM*, 2015.

[12] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proc. of SIGCOMM*, 2015.

[13] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. of SIGCOMM*, 2015.

[14] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of ATC*, 2017.

[15] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief. Lrc: Dependency-aware cache management for data analytics clusters. In *Proc. of INFOCOM*, 2017.

[16] C. Chen, W. Wang, and B. Li. Speculative slot reservation: Enforcing service isolation for dependent data-parallel computations. In *Proc. of ICDCS*, 2017.

[17] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. Parallel real-time scheduling of DAGs. *TPDS*, 25(12), 2014.

[18] Z. Huang, M. Weinberg, L. Zheng, M. Chiang, and C. Joe-Wong. Discovering valuations and enforcing

truthfulness in a deadlineaware scheduler. In *Proc. of INFOCOM*, 2017.

[19] S. Su, Q. Li, J.and Huang, X. Huang, K. Shuang, and J. Wang. Cost-efficient task scheduling for executing large programs in the cloud. *PC*, 39(4), 2013.

[20] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of SoCC*, 2012.

[21] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In *Proc. of SoCC*, 2013.

[22] A. Balasubramanian, A. Sussman, and N. Sadeh. Decentralized preemptive scheduling across heterogeneous multi-core grid resources. In *Proc. of JSSPP*, 2013.

[23] 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. of SoCC*, 2014.

[24] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *Proc. of OSDI*, 2014.

[25] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang. Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. In *Proc. of INFOCOM*, 2013.

[26] M. Alicherry and T.V. Lakshman. Optimizing data access latencies in cloud systems by intelligent virtual machine placement. In *Proc. of INFOCOM*, 2013.

[27] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu. Network scheduling aware task placement in datacenters. In *Proc. of CONEXT*, 2016.

[28] M. A. Vasile, F. Pop, R. I. Tutueanu, V. Cristea, and J. Kołodziej. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *FGCS*, 51, 2015.

[29] S. Niu, J. Zhai, X. Ma, M. Liu, Y. Zhai, W. Chen, and W. Zheng. Employing checkpoint to improve job scheduling in large-scale systems. In *Proc. of JSSPP*, 2012.

[30] F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw-Hill Higher Education, 2005.

[31] CPLEX linear program solver. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/ [accessed in July 2017].

[32] L. E. Schrage and L. W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4), 1996.

[33] N. Megiddo. *On the complexity of linear programming*. 1986.

[34] Palmetto cluster. http://citi.clemson.edu/palmetto/ [accessed in July 2017].

[35] Amazon EC2. http://aws.amazon.com/ec2 [accessed in July 2017].

[36] J. Liu, H. Shen, and H. S. Narman. CCRP: Customized cooperative resource provisioning for high resource utilization in clouds. In *Proc. of IEEE Big Data*, pages 243–252, Washington D.C., 2016.

[37] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-agnostic flow scheduling for commodity data centers. In *Proc. of NSDI*, 2015.

[38] Google trace. https://code.google.com/p/googleclusterdata/ [accessed in July 2017].

[39] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proc. of FAST*, 2012.

[40] A. L. Shimpi. The SSD anthology: Understanding SSDs and new drives from OCZ, 2014.

[41] J. Liu, H. Shen, and L. Chen. CORP: Cooperative opportunistic resource provisioning for short-lived jobs in cloud systems. In *Proc. of IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[42] M. Bertogna and S. Baruah. Limited preemption edf scheduling of sporadic task systems. *IEEE Trans. on II*, 6(4), 2010.

[43] J. Liu and H. Shen. A low-cost multi-failure resilient replication scheme for high data availability in cloud storage. In *Proc. of HiPC*, pages 242–251, 2016.