# Do the Hard Stuff First: Scheduling Dependent Computations in Data-Analytics Clusters

Robert Grandl‡, Srikanth Kandula, Sriram Rao, Aditya Akella‡, Janardhan Kulkarni
Microsoft and University of Wisconsin-Madison‡
MSR-TR-2016-19

**Abstract**– We present DagPS, a scheduler that improves cluster utilization and job completion times by packing tasks with multi-resource requirements and inter-dependencies. While the underlying scheduling problem is intractable in general, DagPS is nearly optimal on the job DAGs that appear in production clusters at a large enterprise. Our key insight is that carefully handling the long-running tasks and those with tough-to-pack resource requirements will lead to good schedules for DAGs. However, which subset of tasks to treat carefully is *a priori* unclear. DagPS offers a novel search procedure that evaluates various possibilities and outputs a valid schedule. An online component enforces the schedules desired by the various jobs running on the cluster. In addition, it packs tasks and, for any desired fairness scheme, guarantees bounded unfairness. We evaluate DagPS on a 200 server cluster using traces of over 20,000 DAGs collected from a large production cluster. Relative to the state-of-the art schedulers, DagPS speeds up half of the jobs by over 30%.

## 1. INTRODUCTION

DAGs (directed acyclic graphs) are a powerfully general abstraction for scheduling problems. Scheduling network transfers of a multi-way join or the work in a geo-distributed analytics job and many others can be represented as DAGs. However, scheduling even one DAG is known to be an NP-hard problem [52, 53].

Consequently, existing work focuses on special cases of the DAG scheduling problem using simplifying assumptions such as: ignore dependencies, only consider chains, assume only two types of resources or only one machine or that the vertices have similar resource requirements [18, 20, 21, 35, 48, 60, 66]. However, the assumptions that underlie these approaches often do not hold in practical settings, motivating us to take a fresh look at this problem.

We illustrate the challenges in the context of job DAGs in data-analytics clusters. Here, each DAG vertex represents a computational task and edges encode input-output dependencies. Programming models such as SparkSQL, Dryad and Tez [3, 19, 42] lead to job DAGs that violate many of the above assumptions. Traces from a large cluster reveal that (a) DAGs have complex structures with the median job having a depth of seven and a thousand tasks, (b) there is substantial variation in compute, memory, network and disk usages across tasks (stdev./avg in requirements is nearly 1), (c) task runtimes range from sub-second to hundreds of seconds, and (d) clusters suffer from resource fragmentation across machines. The net effect of these challenges, based on our analysis, is that the completion times of jobs in this production cluster can be improved by 50% for half the DAGs.

The problem is important because data-analytics clusters run thousands of mission critical jobs each day in enterprises and in the cloud [1, 10]. Even modest improvements in job throughput significantly improves the ROI (return-on-investment) of these clusters; and quicker job completion reduces the lag between data collection and decisions (i.e., "time to insight") which potentially increases revenue [61].

To identify a good schedule for one DAG, we observe that the pathologically bad schedules in today's approaches mostly arise due to these reasons: (a) long-running tasks have no other work to overlap with them and (b) the tasks that are runnable do not pack well with each other. Our core idea, in response, is rather simple: identify the potentially *troublesome* tasks, such as those that run for a very long time or are hard to pack, and place them first on a *virtual resource-time space*. This space would have $d + 1$ dimensions when tasks require $d$ resources; the last dimension being time. Our claim is that placing the troublesome tasks first leads to a good schedule since the remaining tasks can be placed into resultant holes in this space.

Unfortunately, scheduling one DAG well does not suffice. Production cluster schedulers have many concurrent jobs, online arrivals and short-lived tasks [8, 38, 56, 62]. Together, these impose a strict time-budget on scheduling. Also, sharing criteria such as fairness have to be addressed during scheduling. Hence, production clusters are forced to use simple, online heuristics.

We ask whether it is possible to efficiently schedule complex DAGs *while* retaining the advantageous properties of today's production schedulers such as reacting in an online manner, considering multiple objectives etc.

To this end, we design a new cluster scheduler DagPS. At job submission time or soon thereafter, DagPS builds a preferred schedule for a single job DAG by placing the troublesome tasks first. DagPS solves two key challenges in realizing this idea: (1) the best choice of troublesome tasks is intractable to compute and (2) *dead-ends* may arise because tasks are placed out-of-order (e.g., troublesome go first) and it is apriori unclear how much slack space should be set aside. DagPS employs a performant search procedure to address the first challenge and has a placement procedure that provably avoids dead-ends for the second challenge. Figure 1 shows an

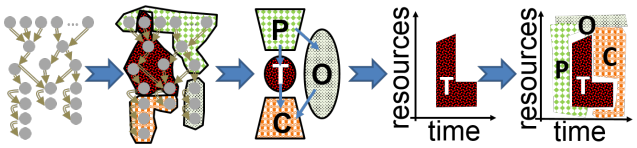



**Figure 1: Shows steps taken by DAGPS from a DAG on the left to its schedule on the right. Troublesome tasks T (in red) are placed first. The remaining tasks (parents P, children C and other O) are placed *on top* of T in a careful order to ensure compactness and respect dependencies.**

example.

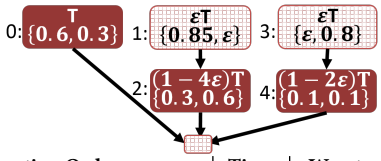
□e schedules constructed for each DAG are passed on to a second online component of DagPS which coordinates between the various DAGs running in the cluster and also reconciles between their multiple, potentially discordant, objectives. For example, a fairness scheme such as DRF may require a certain job to get resources next, but multi-resource packing–which we use to reduce resource fragmentation–or the preferred schedules above may indicate that some other task should be picked next. Our reconciliation heuristic, colloquially, attempts to *follow the majority*; that is it can violate an objective, say fairness, when multiple other objectives counterweight it. However, to maintain predictable performance, our reconciliation heuristic limits maximum unfairness to an operator-con□gured threshold.

We have implemented the two components of DagPS in Apache YARN and Tez and have experimented with jobs from TPC-DS, TPC-H and other benchmarks on a 200 server cluster. Further, we also evaluate DagPS in simulations on 20,000 DAGs from a production cluster.

To summarize, we make theoretical as well as practical contributions in this work. Our key contributions are:

- A characterization of the DAGs seen in production at a large enterprise and an analysis of the performance of various DAG scheduling algorithms (§2).
- A novel DAG scheduler that combines multi-resource packing and dependency awareness (§4).
- An online scheduler that mimics the preferred schedules for all the jobs on the cluster while bounding unfairness (§5) for many models of fairness [7, 13, 33].
- A new lower bound on the completion time of a DAG (§6). Using this we show that the schedules built by DagPS's o□ine component are within 1.04 times OPT for half of the production DAGs; three quarters are within 1.13 times and the worst is 1.75 times OPT.
- An implementation that we intend to release as open source (§7).
- Our experiments show that DagPS improves the completion time of half of the DAGs by 19 – 31%; the number varies across benchmarks. □e improvement for production DAGs is at the high end of the range because these DAGs are more complex and have diverse resource demands.

Lastly, while our work is presented in the context of cluster scheduling, as noted above, similar DAG scheduling problems arise in other domains. We o□er early results in Section 9 from applying DagPS to scheduling DAGs arising in distributed build systems [4, 34] and in request-response work-□ows [45, 67].




| Technique | Execution Order | Time | Worst-case |
|---|---|---|---|
| OPT | $\{t_1, t_3\} \to \{t_0, t_2, t_4\} \to$ | $T$ | – |
| CPSched | $t_0 \to t_3 \to t_4 \to t_1 \to t_2 \to$ | $3T$ | $O(n) \times$ OPT |
| Tetris | $t_0 \to t_1 \to t_2 \to t_3 \to t_4 \to$ | $3T$ | $O(d) \times$ OPT |

**Figure 2: An example DAG where Tetris [37] and Critical Path Scheduling take 3× longer than the optimal algo OPT. Here, DAGPS equals OPT. Details are in §2.2. Assume $\varepsilon \to 0$.**

## 2. PRIMER ON SCHEDULING JOB DAGS

### 2.1 Problem definition

Let each job be represented as a directed acyclic graph $\mathcal{G} = \{V, E\}$. Each node in $V$ is a task with demands for various resources. Edges in $E$ encode precedence constraints between tasks. Many jobs can simultaneously run in a cluster. □e cluster is a group of servers organized as per some network topology.

DagPS considers task demands along four resource dimensions (cores, memory, disk and network bandwidth). Depending on placement, tasks may need resources at more than one machine (e.g., if input is remote) or along network paths. □e network bottlenecks are near the edges (at the source or destination servers and top-of-rack switches) in today's datacenter topologies [14, 16, 65, 59]. Some systems require users to specify the DAG $\mathcal{G}$ explicitly [2, 32, 74] whereas others use a query optimizer to generate $\mathcal{G}$ [24]. Production schedulers already allow users to specify task demands (e.g., [1 core, 1 GB] is the default for tasks in Hadoop 2.6). Note that such annotation tends to be incomplete (network and disk usage is not speci□able) and is in practice signi□cantly overestimated since tasks that exceed their speci□ed usage will be killed. Similar to other reports [15, 23, 46], up to 40% of the jobs in the examined cluster are *recurring*. For such jobs, DagPS uses past job history to estimate task runtimes and resource needs. For the remaining ad-hoc jobs, DagPS uses pro□les from similar jobs and adapts these pro□les online (see §7).

Given a set of concurrent jobs $\{\mathcal{G}\}$, the cluster scheduler maps tasks on to machines while meeting resource capacity limits and dependencies between tasks. Improving **performance**—measured in terms of the job throughput (or makespan) and the average job completion time—is crucial, while also maintaining **fairness**—measured in terms of how resources are divided amongst groups of jobs per some requirement (e.g., DRF or slot-fairness).

### 2.2 An illustrative example

We use the DAG shown in Figure 2 to illustrate the scheduling issues. Each node represents one task: the node

labels represent the task duration (top) and the demands for two resources (bottom). Assume that the total resource available is 1 for both resources and let $\varepsilon$ represent a small value.

Intuitively, a good schedule would overlap the long-running tasks shown with a dark background. The resulting optimal schedule (OPT) is shown in the table (see Figure 2). OPT overlaps the execution of all the long-running tasks– $t_0$, $t_2$ and $t_4$– and finishes in $T$. However, such long-running/resource intensive tasks can be present anywhere in the DAG, and it is unlikely that greedy local schedulers can overlap these tasks. To compare, the table also shows the schedules generated by a typical DAG scheduler, and a state-of-the-art *packer* which carefully packs tasks onto machines to maximize resource utilization. We discuss them next.

DAG schedulers such as critical path scheduling (CPSched) pick tasks along the critical path (CP) in the DAG. The CP for a task is the longest path from the task to the job output. The figure also shows the task execution order with CPSched.[1] CPSched ignores the resources needed by tasks and does not pack. Consequently, for this example, CPSched performs poorly because it does not schedule tasks that are not on the critical path first (such as $t_1$, $t_3$) even though doing so reduces resource fragmentation by overlapping the long-running tasks.

On the other hand, packers such as, Tetris [37], pack tasks to machines by matching along multiple resource dimensions. Tetris greedily picks the task with the highest value of the dot product between task's demand vector and the available resource vector. The figure also shows the task execution order with Tetris.[2] Tetris does not account for dependencies. Its packing heuristic only considers the tasks that are currently schedulable. In this example, Tetris performs poorly because it will not choose locally inferior packing options (such as running $t_1$ instead of $t_0$) even when doing so can lead to a better global packing.

DagPS achieves the optimal schedule for this example. When searching for troublesome subsets, it will consider the subset $\{t_0, t_2, t_4\}$ because these tasks run for much longer. As shown in Figure 1, the troublesome tasks will be placed first. Since there are no dependencies among them, they will run at the same time. The parents ($\{t_1, t_3\}$) and any children are then placed *on top*; i.e., compactly before and after the troublesome tasks.

## 2.3 Analyzing DAGs in Production

We examined the production jobs from a cluster of tens of thousands of servers at a large enterprise. We also analyzed jobs from a 200 server cluster that ran Hive [68] jobs and jobs from a high performance computing cluster [6].

To quantify potential gains, we compare the runtime of the DAGs in production to three measures. The first measure, CPLength is the duration of the DAG's critical path. If the available parallelism is infinite, the DAG would finish within CPLength. The second measure, TWork, is the total work in the DAG normalized by the cluster share of that DAG. If there were no dependencies and perfect packing, a DAG would finish within TWork. In practice, both of these measures are quite loose– the first ignores all the work off the critical path and the second ignores dependencies. Hence, our third measure is a new improved lower bound NewLB that uses the specific structure of data-parallel DAGs. Further details are in §6 but intuitively NewLB leverages the fact that unlike random DAGs, all the tasks in a job stage (e.g., a map or reduce or join) have similar dependencies, durations and resource needs.

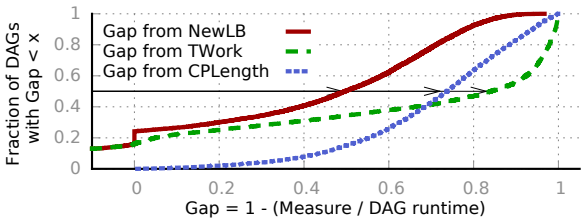



**Figure 3: CDF of *gap* between DAG runtime and several measures. Gap is computed as $1 - \frac{\text{measure}}{\text{DAG runtime}}$.**

| | CPU | Mem. | Network | | Disk | |
|---|---|---|---|---|---|---|
| | | | Read | Write | Read | Write |
| Enterprise: Private Stack | 0.76 | 1.01 | 1.69 | 7.08 | 1.39 | 1.94 |
| Enterprise: Hive | 0.89 | 0.42 | 0.77 | 1.34 | 1.59 | 1.41 |
| HPC: Condor | 0.53 | 0.80 | N/A | N/A | 1.55 (R+W) | |

**Table 1: Coefficient-of-variation (= stdev./avg.) of tasks' demands for various resource. Across three examined frameworks, tasks exhibit substantial variability (CoV ~ 1) for many resources.**

Figure 3 plots a CDF of the gap over all DAGs for these three measures. Observe that half of the jobs have a gap of over 70% for both CPLength and TWork. The gap relative to NewLB is smaller, indicating that the newer bound is tighter, but the gap is still over 50% for half of the jobs. That is, they take over two times longer than they could.

A few issues are worth noting for this result. First, some DAGs finish faster than their TWork and NewLB measures. This is because our production scheduler is work conserving and can give jobs more than their fair share. Second, we know that jobs take longer in production because of runtime artifacts such as task failures or stragglers [17, 50]. What fraction of the gap is explained due to these reasons? When computing the job completion times to use in this result, we attempted to explicitly avoid these issues as follows. First, we chose the fastest completion time from among groups of related recurring jobs. It is unlikely that every execution suffers from failures. Second, we shorten the completion time of a job by deducting all periods when the job has fewer than 10 tasks running concurrently. This is explicitly corrects for stragglers–one or a few tasks holding up job progress. Hence, we believe that the remaining gap is likely due to the scheduler's inability to pack tasks with dependencies.

To understand the causes for the performance gap further, we characterize the DAGs along the following dimensions:

[1] CP of $t_0$, $t_1$, $t_3$ is $T$, $T(1-3\varepsilon)$ and $T(1-\varepsilon)$ respectively. The demands of these tasks ensure that they cannot run simultaneously.

[2] Tetris' packing score for each task, in descending order, is $t_0$=$t_2$=0.9, $t_1$=0.85, $t_3$=0.8 and $t_4$=0.2.

| "Work" that is … | Percentage of total work in the DAG | | | | |
|---|---|---|---|---|---|
| | [0-20) | [20-40) | [40-60) | [60-80) | [80-100] |
| on CriticalPath | 14.6% | 13.2% | 15.2% | 15.3% | **41.6%** |
| "unconstrained" | 15.6% | 20.4% | 14.2% | 16.4% | **33.3%** |
| "unordered" | 0 | 3.6% | 11.8% | 27.9% | **56.6%** |

**Table 2: Bucketed histogram of where the work lies in DAGs. Each entry denotes the fraction of all DAGs that have the metric labeled on the row in the range denoted by the column. For example, 14.6% of DAGs have [0, 20)% of their total work on the critical path.**

**What do the DAGs look like?** By depth, we refer to the number of tasks on the critical path. A map-reduce job has depth 2. We find that the median DAG has depth 7. Further, we find that the median (75th percentile) task in-degree and out-degree are 7 (48) and 1 (4) respectively. If DAGs are chains of tasks, in- and out-degree's will be 1. A more detailed characterization of DAGs including tree widths and path widths has been omitted for brevity. Our summary is that the vast majority of DAGs have complex structures.

**How diverse are the resource demands of tasks?** Table 1 shows the coefficient-of-variation (CoV) across tasks for various resources. We find that the resource demands vary substantially. The variability is possibly due to differences in work at each task: some are compute heavy (e.g., user-defined code that processes videos) whereas other tasks are memory heavy (e.g., in-memory sorts).

**Where does the work lie in a DAG?** We now focus on the more important parts of each DAG– the tasks that do more work (measured as the product of task duration and resource needs). Let `CPWork` be the total work in the tasks that lie on the critical path. From Table 2, 42% of DAGs have `CPWork` above 80%. DAG-aware schedulers may do well for such DAGs. Let `UnconstrainedWork` be the total work in tasks with no parents (i.e., no dependencies). We see that roughly 33% of the DAGs have `UnconstrainedWork` above 80%. Such DAGs will benefit from packers. The above cases are not mutually exclusive and together account for 54% of DAGs. For the other 46% of DAGs, neither packers nor criticality-based schedulers may work well.

Let `MaxUnorderedWork` be the largest work in a set of tasks that are neither parents nor children of each other. Table 2 shows that 57% of DAGs have `MaxUnorderedWork` above 80%. That is, if ancestors of the unordered tasks were scheduled appropriately, substantial gains can accrue from packing the tasks in the maximal unordered subset.

From the above analysis, we observe that (1) production jobs have large DAGs that are neither a bunch of unrelated stages nor a chain of stages, and (2) a packing+dependency-aware scheduler can offer substantial improvements.

## 2.4 Analytical Results

We take a step back to offer some more general comments. First, DAG schedulers have to be aware of dependencies. That is, considering just the runnable tasks does not suffice.

**Lemma 1.** Any scheduling algorithm, deterministic or randomized, that does not account for the DAG structure is at least $\Omega(d)$ times `OPT` where $d$ is the number of resources.

For deterministic algorithms, the proof follows from designing an adversarial DAG for any scheduler. We extend to randomized algorithms by using Yao's max-min principle (see A). Lemma 1 applies to all multi-resource packers [37, 57, 71, 72] since they ignore dependencies.

Second, and less formally, we note that schedulers have to be aware of resource heterogeneity. Many known scheduling algorithms have poor worst-case performance. In particular:

**Lemma 2.** Critical path scheduling can be $\Omega(n)$ times `OPT` where $n$ is the number of tasks in a DAG and Tetris can be $(2d - 2)$ times `OPT`.

The proof is by designing adversarial DAGs for each scheduler (see B).

To place these results in context, note that $d$ is about 4 (cores, memory, network, disk) and can be larger when tasks require resources at other servers or on many network links. Further, the median DAG has hundreds of tasks ($n$). DagPS is close to `OPT` on all of the described examples. Furthermore, DagPS is within 1.04 times optimal for half of the production DAGs (estimated using our new lower bound).

Finally, we note the following:

**Lemma 3.** If there were no precedence constraints and tasks were malleable, `OPT` is achievable by a greedy algorithm.

We say a task is malleable if assigning any (non-negative) portion of its demand $p$ will cause it to make progress at rate $p$. In particular, tasks can be paused ($p = 0$) at any time which is also referred to as tasks being preemptible. The proof follows by describing the simple greedy algorithm which we omit here for brevity.

Our summary is that practical DAGs are hard to schedule because of their complex structure as well as discretization issues when tasks need multiple resources (fragmentation, task placement etc.)

# 3. NOVEL IDEAS IN DagPS

Cluster scheduling is the problem of matching tasks to machines. Every practical scheduler today does so in an online manner but has very tight timing constraints since clusters have thousands of servers, many jobs each having many pending tasks and tasks that finish in seconds or less [74, 8]. Given such stringent time budget, carefully considering large DAGs seems hopeless.

As noted in §1, a key design decision in DagPS is to divide this problem into two parts. An offline component constructs careful schedules for a single DAG. We call these the *preferred schedules*. A second online component enforces the preferred schedules of the various jobs running in the cluster. We elaborate on each of these parts below. Figure 4 shows an example of how the two parts may inter-operate in a YARN-style architecture. Dividing a complex problem into parts and independently solving each part often leads to a sub-optimal solution. Unfortunately, we have no guarantees for our particular division. However, it can scale to large clusters and outperforms the state-of-art in experiments.

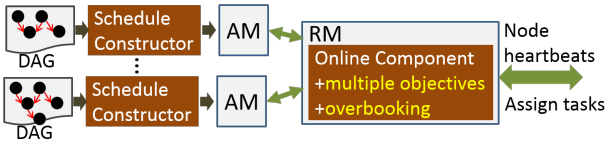



**Figure 4: DagPS builds schedules per DAG at job submission. The runtime component handles online aspects. AM and RM refer to the YARN's application and resource manager components.**

To find a compact schedule for a single DAG, our idea is to place the troublesome tasks, i.e., those that can lead to a poor schedule, first onto a virtual space. Intuitively, this maximizes the likelihood that any *holes*, un-used parts of the resource-time space, can be filled by other tasks. However, finding the best choice of troublesome tasks is as hard as finding a good schedule for the DAG. We use an efficient search strategy that mimics dynamic programming: it picks subsets that are more likely to be useful and avoids redundant exploration. Further, placing troublesome tasks first can lead to *dead-ends*. We define dead-end to be an arrangement of a subset of the DAG in the virtual space on which the remaining tasks cannot be placed without violating dependencies. Our strategy is to divide the DAG into subsets of tasks and place one subset at a time. While intra-subset dependencies are trivially handled by schedule construction, inter-subset dependencies are handled by restricting the order in which the various subsets are placed. We prove that the resultant placement has no *dead-ends*.

The online component has to co-ordinate between some potentially discordant directives. Each job running in the cluster offers a preferred schedule for its tasks (constructed as above). Fairness models such as DRF may dictate which job (or queue) should be served next. The set of tasks that is advantageous for packing (e.g., maximal use of multiple resources) can be different from both the above choices. We offer a simple method to reconcile these various directives. Our idea is to compute a real-valued score for each pending task that incorporates the above aspects *softly*. That is, the score trades-off violations on some directives if the other directives weigh strongly against it. For example, we can pick a task that is less useful from a packing perspective if it appears much earlier on the preferred schedule. Two key novel aspects are judiciously overbooking resources and bounding the extent of unfairness. Overbooking allows schedules that overload a machine or a network link if the cost of doing so (slowing-down of all tasks using that resource) is less than the benefit (can finish more tasks).

The offline component of DagPS is described next; the online component is described in Section 5.

# 4. SCHEDULING ONE DAG

DagPS builds the schedule for a DAG in three steps. Figure 1 illustrates these steps and Figure 5 has a simplified pseudocode. First, DagPS identifies some troublesome tasks and divides the DAG into four subsets (§4.1). Second, tasks in a subset are packed greedily onto the virtual space while respecting dependencies (§4.2). Third, DagPS carefully restricts

**Definitions:** In DAG $\mathcal{G}$, $t$ denotes a task and $s$ denotes a stage, i.e., a collection of similar tasks.
Let $\mathcal{V}$ denote all the stages (and hence the tasks) in $\mathcal{G}$.
Let $\mathcal{C}(s,\mathcal{G}), \mathcal{P}(s,\mathcal{G}), \mathcal{D}(s,\mathcal{G}), \mathcal{A}(s,\mathcal{G}), \mathcal{U}(s,\mathcal{G})$ denote the children, parents, descendants, ancestors and unordered neighbors of $s$ in $\mathcal{G}$.
For clarity, $\mathcal{U}(s,\mathcal{G}) = \mathcal{V} - \mathcal{A}(s,\mathcal{G}) - \mathcal{D}(s,\mathcal{G}) - \{s\}$

```
1  Func: BuildSchedule:
2  Input: G: a DAG, m: number of machines
3  Output: An ordered list of tasks t ∈ G
4  S_best ← ∅// best schedule for G thus far
5  foreach sets {T, O, P, C} ∈ CandidateTroublesomeTasks(G) do
6      Space S ← CreateSpace(m) //resource-time space
7      S ← PlaceTasks(T, S, G)// trouble goes first
8      S ← TrySubsetOrders({OCP, OPC, COP, POC}, S, G)
9      if S < S_best then S_best ← S //keep the best schedule;
10 return OrderTasks(G, S_best)
```

**Figure 5: Pseudocode for constructing the schedule for a DAG. Helper methods are in Figure 6.**

the order in which different subsets are placed such that the troublesome tasks go first and there are no dead-ends (§4.3). DagPS picks the most compact schedule after iterating over many choices for troublesome tasks. We discuss some enhancements in §4.4. The resulting schedule is passed on to the online component (§5).

## 4.1 Searching for troublesome tasks

To identify *troublesome* tasks, DagPS computes two scores per task. The first, `LongScore`, divides the task duration by the maximum value across all tasks. Tasks with a higher score are more likely to be on the critical path and can benefit from being placed first because other work can overlap with them. The second, `FragScore`, reflects the packability of tasks in a stage (e.g., a map or a reduce). It is computed by dividing the total work in a stage (`TWork` defined in §2.2) by how long a greedy packer will take to schedule that stage. Tasks that are more difficult to pack would have a lower `FragScore`. Given thresholds $l$ and $f$, DagPS picks tasks with `LongScore` $\geq l$ or `FragScore` $\leq f$. Intuitively, doing so biases towards selecting tasks that are more likely to hurt the schedule because they are too long or too difficult to pack. DagPS iterates over different values for the $l$ and $f$ thresholds to find a compact schedule.

To speed up this search, (1) rather than choose the threshold values arbitrarily, DagPS picks values that are discriminative, i.e., those that allow different subsets of tasks to be considered as troublesome and (2) DagPS remembers the set of troublesome tasks that were already explored (by previous settings of the thresholds) so that it will construct a schedule only once per unique troublesome set.

As shown in Figure 6, the set T is a closure over the chosen troublesome tasks. That is, T contains the troublesome tasks and all tasks that lie on a path in the DAG between two troublesome tasks. The parent and child subsets P, C consist of tasks that are not in T but have a descendant or ancestor in T respectively. The subset O consists of the remaining tasks.

## 4.2 Compactly placing tasks

Given a subset of tasks and a partially occupied space, how

See Definitions atop Fig. 5.

```
Func: CandidateTroublesomeTasks:
Input: DAG 𝒢; Output: list ℒ of sets T, O, P, C
// choose a candidate set of troublesome tasks; per choice, divide 𝒢 into four sets
ℒ ← ∅
∀v ∈ 𝒢, LongScore(v) ← v.duration/max_{v'∈𝒢} v'.duration
∀v ∈ 𝒢, v in stage s, FragScore(v) ← TWork(s)/ExecutionTime(s)
foreach l ∈ δ, 2δ, ...1 do
    foreach f ∈ δ, 2δ, ...1 do
        T ← {v ∈ 𝒢 | LongScore(v) ≥ l or FragScore(v) ≤ f}
        T ← Closure(T)
        if T ∈ ℒ then continue // ignore duplicates;
        P ← ⋃_{v∈T} 𝒜(v, 𝒢); C ← ⋃_{v∈T} 𝒟(v, 𝒢);
        ℒ ← ℒ ∪ {T, 𝒱 − T − P − C, P, C}
```

**Figure 6: Identifying various candidates for troublesome tasks and dividing the DAG into four subsets.**

best to pack the tasks while respecting dependencies? One can choose to place the parents first or the children first. We call these the *forward* and *backward* placements respectively. More formally, the forward placement recursively picks a task all of whose ancestors have already been placed on the space and puts it at the earliest possible time after its latest finishing ancestor. The backward placement is analogously defined. Intuitively, both placements respect dependencies but can lead to very different schedules since greedy packing yields different results based on which tasks are placed first. Figure 7:`PlaceTasksF` shows one way to do this. Traversing the tasks in either placement has $n \log n$ complexity for a subset of $n$ tasks and if there are $m$ machines, placing tasks greedily has $n \log(mn)$ complexity.

## 4.3 Subset orders that guarantee feasibility

For each division of DAG into subsets T, O, P, C, DagPS considers these four orders: TOCP, TOPC, TPOC or TCOP. That is, in the TOCP order, it first places all tasks in T, then tasks in O, then tasks in C and finally all tasks in P. Intuitively, this helps because the troublesome subset T is always placed first. Further, we will shortly prove that these are the only orders beginning with T that will avoid *dead-ends*.

A subtle issue is worth discussing. Only one of the forwards or backwards placements (described above in §4.2) are appropriate for some subsets of tasks. For example, tasks in P cannot be placed *forwards* since some descendants of these tasks may already have been placed (such as those in T). As we saw above, the forwards placement places a task after its last finishing ancestor but ignores descendants and can hence violate dependencies if used for P. Analogously, tasks in C cannot be placed *backwards*. Tasks in O can be placed in one or both placements, depending on the inter-subset order. Finally, since the tasks in T are placed onto an empty space they can be placed either forwards or backwards. Formally, this logic is encoded in Figure 7:`TrySubsetOrders`. We prove the following lemma.

**Lemma 4.** (Correctness) The method described in §4.1–§4.3 satisfies all dependencies and is free of *dead-ends*. (Completeness) Further, the method explores every order that places troublesome tasks first and is free of *dead-ends*.

```
Func: PlaceTasksF: // forward placement
Inputs: 𝒱: subset of tasks to be placed, 𝒮: space (partially filled), 𝒢: a DAG
Output: a new space with tasks in 𝒱 placed atop 𝒮
𝒮 ← Clone(𝒮)
finished placement set ℱ ← {v ∈ 𝒢 | v already placed in 𝒮}
while true do
    ready set ℛ ← {v ∈ 𝒱 − ℱ | 𝒫(v, 𝒢) already placed in 𝒮}
    if ℛ = ∅ then break // all done;
    v' ← task in ℛ with longest runtime
    t ← max_{v∈𝒫(v,𝒢)} EndTime(v, 𝒮)
    // place v' at earliest time ≥ t when its resource needs can be met
    ℱ ← ℱ ∪ v'
Func: PlaceTasks(𝒱, 𝒮, 𝒢):// inputs and output are same as PlaceTasksF
return min (PlaceTasksF(𝒱, 𝒮, 𝒢), PlaceTasksB(𝒱, 𝒮, 𝒢))
Func: PlaceTasksB: // only backwards, analogous to PlaceTasksF.
Func: TrySubsetOrders:
Input: 𝒢: a DAG, 𝒮_in: space with tasks in T already placed
Output: Most compact placement of all tasks.
𝒮1, 𝒮2, 𝒮3, 𝒮4 ← Clone(𝒮_in)
return min( // pick the most compact among all feasible orders
PlaceTasksF(C, PlaceTasksB(P, (PlaceTasks(O, 𝒮1, 𝒢)), 𝒢), 𝒢),// OPC
PlaceTasksB(P, PlaceTasksF(C, (PlaceTasks(O, 𝒮2, 𝒢)), 𝒢), 𝒢),// OCP
PlaceTasksB(P, PlaceTasksB(O, (PlaceTasksF(C, 𝒮3, 𝒢)), 𝒢), 𝒢),// COP
PlaceTasksF(C, PlaceTasksF(O, (PlaceTasksB(P, 𝒮4, 𝒢)), 𝒢), 𝒢)// POC
);
```

**Figure 7: Pseudocode for the functions described in §4.2 and §4.3.**

We omit a detailed proof due to space constraints. Intuitively however, the proof follows from (1) all four subsets are closed and hence intra-subset dependencies are respected by both the placements in §4.2, (2) the inter-subset orders and the corresponding restrictions to only use forwards and/or backwards placements specified in §4.3 ensure dependencies across subsets are respected and finally, (3) every other order that begins with T either violates dependencies or leads to a dead-end (e.g., in TPCO, placing tasks in O can dead-end because some ancestors and descendants have already been placed).

## 4.4 Enhancements

We note a few enhancements. First, due to barriers it is possible to partition a DAG into parts that are totally ordered. Hence, any schedule for the DAG is a concatenation of per-partition schedules. This lowers complexity because one execution of `BuildSchedule` will be replaced by several executions each having fewer tasks. 24% of the production DAGs can be split into four or more parts. Second, and along similar lines, whenever possible we reduce complexity by reasoning over stages. Stages are collections of tasks and are 10 to $10^3$ times fewer in number than tasks. Finally, we carefully choose our data-structures (e.g., a time and resource indexed hash map of free regions in space) so that the most frequent operation, picking a region in resource-time space where a task will fit as described in §4.2, can be executed efficiently.

# 5. SCHEDULING MANY DAGS

We describe our online algorithm that matches tasks to machines while co-ordinating discordant objectives: fairness, packing and enforcing the per-DAG schedules built by §4. We offer the pseudocode in Figure 8 for complete-

**Func:** FindAppropriateTasksForMachine:
**Input:** $\mathbf{m}$: vector of available resources at machine; $\mathcal{J}$: set of jobs with task details$\{t_{\text{duration}}, \mathbf{t}_{\text{demands}}, t_{\text{priScore}}\}$; `deficit`: counters for fairness;
**Parameters:** $\kappa$: unfairness bound; `rp`: remote penalty
**Output:** $\mathcal{S}$, the set of tasks to be allocated on the machine
$\mathcal{S} \leftarrow \emptyset$
**while** *true* **do**
  **foreach** *task* $t$ **do**
    $\{\texttt{pScore}_t, \texttt{oScore}_t\} \leftarrow \{0, 0\}$
    $\texttt{rPenalty}_t \leftarrow t$ is locality sensitive ? `rp` : 1
    **if** $\mathbf{t}_{demands} \leq \mathbf{m}$ *// fits?* **then**
      $\texttt{pScore}_t \leftarrow (\mathbf{m} \cdot \mathbf{t}_{\text{demands}})\ \texttt{rPenalty}_t$ // dot product
    **else**
      compute $\texttt{oScore}_t$ // overbooking score omitted for brevity.
    job $j \ni t$, $\texttt{srpt}_j \leftarrow \sum_{\text{pending } u \in j} u_{\text{duration}} * |\mathbf{u}_{\text{demands}}|$
    $\texttt{perfScore}_t \leftarrow t_{\text{priScore}}\{\texttt{pScore}_t, \texttt{oScore}_t\} - \eta\texttt{srpt}_j$
  $t^{\text{best}} \leftarrow \arg\max\{\texttt{perfScore}_t | t\}$ // task with highest perf score
  **if** $t^{best} = \emptyset$ **then break** // no new task can be scheduled on this machine;
  $g' \leftarrow$ jobgroup with highest deficit counter
  **if** $\texttt{deficit}_{g'} \geq \kappa C$ **then** $t^{\text{best}} \leftarrow \arg\max\{\texttt{perfScore}_t | t \in g'\}$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup t^{\text{best}}$
  // detail: reduce available resources $\mathbf{m}$.
  $\texttt{deficit}_g \leftarrow \texttt{deficit}_g +$
  $f(\mathbf{t}^{\text{best}}_{\text{demands}}) * \begin{cases} \texttt{fairShare}_g - 1 & t \in \text{jobgroup } g \\ \texttt{fairShare}_g & \text{otherwise} \end{cases}$

**Figure 8: Simplified pseudocode for the online component.**

ness but focus only on (1) how the various objectives are coordinated and (2) how unfairness is bounded.

The pseudocode shows how various individual objectives are estimated. Packing score per task $\texttt{pScore}_t$ is a dot product between task demands and available resources [37]. Using remote resources un-necessarily, for example by scheduling a locality-sensitive task [51] at another machine, is penalized by the value $\texttt{rPenalty}_t$. The value $\texttt{srpt}_j$ estimates the remaining work in a job and is used to prefer short jobs which lowers average job completion time. We claim no novelty thus far. Suppose that $t_{\texttt{priScore}}$ is the order over tasks required by the schedule from §4; $t_{\texttt{priScore}}$ is computed by ranking tasks in increasing order of their begin time and then dividing the rank by the number of tasks in the DAG so that the value is between 1 (task that begins first) and 0 (for the last task).

An initial combination of the above goals happens in the computation of $\texttt{perfScore}_t$. See the first box in Figure 8. A task will have non-zero $\texttt{pScore}_t$ only if its demands fit within available resources. Else, it can have a non-zero $\texttt{oScore}_t$ if it is worth overbooking. We use a lexicographic ordering between these two values. That is, tasks with non-zero `pScore` beat any value of `oScore`. Multiplying with $t_{\texttt{priScore}}$ steers the search towards tasks earlier in the constructed schedule. Finally, $\eta$ is a parameter that is automatically updated based on the average `srpt` and `pScore`. Subtracting $\eta \cdot \texttt{srpt}_j$ prefers shorter jobs. Intuitively, the combined value $\texttt{perfScore}_t$ softly enforces the various objectives. For example, if some task is preferred by all individual objectives (belongs to shortest job, is most packable, is next in the preferred schedule), then it will have the highest `perfScore`. When the objectives are discordant, colloquially, the task preferred

$$\texttt{CPLen}_{\mathcal{G}} = \max_{\text{path } p \in \mathcal{G}} \sum_{\text{task } t \in p} t_{\text{duration}} \tag{1a}$$

$$\texttt{TWork}_{\mathcal{G}} = \max_{\text{resource } r} \frac{1}{C_r} \sum_{t \in \mathcal{G}} t_{\text{duration}} t^r_{\text{demands}} \tag{1b}$$

$$\texttt{ModCP}_{\mathcal{G}} = \max_{p \in \mathcal{G}} \max_{s \in p} (\max(\texttt{TWork}_s, \texttt{CPLen}_s) + \sum_{s' \in p - \{s\}} \min_{t \in s'} t_{\text{dur.}}) \tag{1c}$$

$$\texttt{NewLB}_{\mathcal{G}} = \sum_{\mathcal{G}' \in \text{Partitions}(\mathcal{G})} \max(\texttt{CPLen}_{\mathcal{G}'}, \texttt{TWork}_{\mathcal{G}'}, \texttt{ModCP}_{\mathcal{G}'}) \tag{1d}$$

**Figure 9: Lower bound formulas for DAG $\mathcal{G}$; $p$, $s$, $t$ denote a path through the DAG, a stage and a task respectively. $C$, here, is the capacity available for this job. We developed `ModCP` and `NewLB`.**

by a majority of objectives will have the highest `perfScore`.

To bound unfairness, we use one additional step. We explicitly measure unfairness using deficit counters [64]. When the maximum unfairness (across jobgroups or queues) is above the specified threshold $\kappa C$, where $C$ is the cluster capacity, DagPS picks only among tasks belonging to the most unfairly treated jobgroup. This is shown in the second box in Figure 8. Otherwise DagPS picks the task with the highest `perfScore`. It is easy to see that this bounds unfairness by $\kappa C$. Further, we can support a variety of fairness schemes by choosing how to change the deficit counter. For example, choosing $f() = 1$ mimics slot fairness (see third box in Figure 8), and $f() =$ demand of the dominant resource mimics DRF [33].

## 6. A NEW LOWER BOUND

We develop a new lower bound on the completion time of a DAG of tasks. As we saw in §2.3, previously known lower bounds are very loose. Since the optimal solution is intractable to compute, without a good lower bound, it is hard to assess the quality of a heuristic solution such as DagPS.

Equations 1a and 1b describe the known bounds: critical path length `CPLen` and total work `TWork`. Equation 1d is (a simpler form of) our new lower bound. At a high level, the new lower bound uses some structural properties of these job DAGs. Recall that DAGs can be split into parts that are totally ordered (§4.4). This lets us pick the best lower bound for each part independently. For a DAG that splits into a chain of tasks followed by a group of independent tasks, we could use `CPLen` of the chain plus the `TWork` of the group. A second idea is that on a path through the DAG, at least one stage has to complete entirely. That is, all of the tasks in some stage and at least one task in each other stage on the path have to complete entirely. This leads us to the $\texttt{ModCP}_{\mathcal{G}}$ formula in Equation 1c where one stage $s$ along any path $p$ is replaced with the total work in that stage. A few other ideas are omitted for brevity.

The take-away is that the new lower bound `NewLB` is much tighter and allows us to show that DagPS is close to `OPT`; since by definition of a lower bound DagPS $\geq$ `OPT` $\geq$ `NewLB`.

## 7. DagPS SYSTEM

We have implemented the runtime component (§5) in the Apache YARN resource manager (RM) and the schedule constructor (§4) in the Apache Tez application master (AM). Our

schedule constructor implementation finishes in tens of seconds on all of the DAGs used in experiments; this is in the same ballpark as the time to compile and query-optimize these DAGs. Further, recurring jobs use previously constructed schedules. Each DAG is managed by an instance of the Tez AM which closely resembles other popular frameworks such as FlumeJava [25] and Dryad [42]. The per-job AMs negotiate with the YARN RM for containers to run the job's tasks; each container is a fixed amount of various resources. As part of implementing DagPS, we expanded the interface between the AM and RM to pass additional information, such as the job's pending work and tasks' demands, duration and preferred order. Due to anonymity considerations, we are unable to share full details of our code release. Here, we describe two key implementation challenges: (a) constructing profiles of tasks' resource demands and duration (§7.1), and (b) efficiently implementing the new online task matching logic (§7.2).

## 7.1 Profiling Tasks' Requirements

We estimate and update the tasks' resource demands and durations as follows. Recurring jobs are fairly common in production clusters (up to 40% [15, 23, 46]), executing periodically on newly arriving data (e.g., updating metrics for a dashboard). For these jobs, DagPS extracts statistics from prior runs. In the absence of prior history, we rely on two aspects of data analytics computations that make it amenable to learn profiles at runtime. (1) Tasks in a stage (e.g., map or reduce) have similar profiles and (2) tasks often run in multiple waves due to capacity limits. DagPS measures the progress and resource usage of tasks at runtime. Using the measurements from in-progress and completed tasks, DagPS refines estimates for the remaining tasks. Our evaluation will demonstrate the effectiveness of this approach.

## 7.2 Efficient Online Matching: Bundling

We have redesigned the online scheduler in YARN that matches machines to tasks. From conversations with Hadoop committers, these code-changes help improve matching efficiency and code readability.

Some background: The matching logic is heartbeat based. When a machine heartbeats to the RM, the allocator (1) picks an appropriate task to allocate to that machine, (2) adjusts its data structures (such as, resorting/rescoring) and (3) repeats these steps until all resources on the node have been allocated or all allocation requests have been satisfied.

As part of this work, we support *bundling* allocations. That is, rather than breaking the loop after finding the first schedulable task, we maintain a set of tasks that can all be potentially scheduled on the machine. This so-called *bundle* allows us to schedule multiple tasks in one iteration, admitting non-greedy choices over multiple tasks. For example, if tasks $t_1, t_2, t_3$ are discovered in that order, it may be better to schedule $t_2$ and $t_3$ together rather than schedule $t_1$ by itself. We refactored the scheduler to support bundling; with configurable choices for (1) which tasks to add to the bundle, (2) when to terminate bundling (e.g. the bundle has a good set of tasks) and (3) which tasks to pick from the bundle.

# 8. EVALUATION

Here, we report results from experiments on a 200 server cluster and extensive simulations using 20,000 DAGs from production clusters. Our key findings are:

(1) In experiments on a large server cluster, relative to `Tez` jobs running on `YARN`, DagPS improves completion time of half of the jobs by 19% to 31% across various benchmarks. A quarter of the jobs improve by 30% to 49%.

(2) On the DAGs from production clusters, schedules constructed by DagPS are faster by 25% for half of the DAGs. A quarter of the DAGs improve by 57%. Further, by comparing with our new lower bound, these schedules are optimal for 40% of the jobs and within 13% of optimal for 75% of the jobs.

As part of the evaluation, we offer detailed comparisons with many alternative schedulers and sensitivity analysis to cluster load and parameter choices. We also provide early results on applying DagPS to DAGs from other domains (§9).

## 8.1 Setup

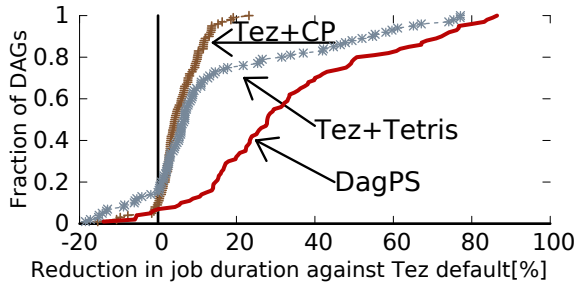
**Our experimental cluster** has 200 servers with two quad-core Intel E2550 processors (hyperthreading enabled), 128 GB RAM, 10 drives, and a 10Gbps network interface. The network has a congestion-free core [14].

**Workload:** Our workload mix consists of jobs from public benchmarks—TPC-H [12], TPC-DS [11], BigBench [5], and jobs from a production cluster that runs Hive jobs (E-Hive). We also use 20K DAGs from a private production system in our simulations. In each experimental run, jobs arrival is modeled via a Poisson process with average inter-arrival time of 25s for 50 minutes. Each job is picked at random from the corresponding benchmark. We built representative inputs and varied input size from GBs to tens of TBs such that the average query completes in a few minutes and the longest finishes in under 10 minutes on the idle cluster. A typical experiment run thus has about 200 jobs and lasts until the last job finishes. The results presented are the median over three runs.

**Compared Schemes:** We experimentally compare DagPS against the following baselines: (1) `Tez` : breadth-first order of tasks in the DAG running atop YARN's Capacity Scheduler (CS), (2) `Tez + CP` : critical path length based order of tasks in the DAG atop CS and (3) `Tez + Tetris` : breadth-first order of tasks in the DAG atop Tetris [37].

Using simulations, we compare DagPS against the following schemes: (4) `BFS` : breadth first order, (5) `CP` : critical path order, (6) `Random` order, (7) `StripPart` [20], (8) `Tetris` [37], and (9) `Coffman – Graham` [30].

All of the above schemes except (7) are work-conserving. (4)–(6) and (8) pick greedily from among the runnable tasks but vary in the specific heuristic. (7) and (9) require more complex schedule construction, as we will discuss later.



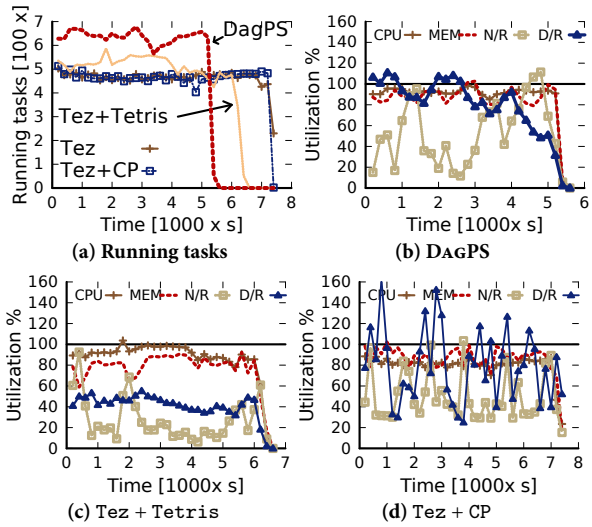

**(a) CDF of gains for jobs on TPC-DS workload**

| Workload | 50$^{th}$ percentile | | | 75$^{th}$ percentile | | |
|---|---|---|---|---|---|---|
| | **D** | T+C | T+T | **D** | T+C | T+T |
| TPC-DS | **27.8** | 4.1 | 6.5 | **45.7** | 8.9 | 16.6 |
| TPC-H | **30.5** | 3.8 | 8.9 | **48.3** | 7.7 | 15.0 |
| BigBench | **25.0** | 6.4 | 6.2 | **33.3** | 21.7 | 18.5 |
| E-Hive | **19.0** | 1.0 | 5.8 | **29.7** | 4.5 | 14.2 |

**D** stands for DagPS. T+C and T+T denote `Tez + CP` and `Tez + Tetris` respectively (see §8.1). The improvements are relative to `Tez`.

**(b) Improvements in job completion time across all the workloads**

**Figure 10: Comparing completion time improvements of various schemes relative to `Tez`.**




**(a) Running tasks** **(b) DagPS**

**(c) `Tez + Tetris`** **(d) `Tez + CP`**

**Figure 11: For a cluster run with 200 jobs, a time lapse of how many tasks are running (leftmost) and how many resources are allocated by each scheme. $N/R$ represents the amount of network read, $D/R$ the disk read and $D/W$ the corresponding disk write.**

**Metrics:** Improvement in job completion time is our key metric. Between two schemes, we measure the *normalized gap* in job completion time. That is, the difference in the runtime achieved for the same job divided by the runtime of the job with some scheme; the normalization lets us compare across jobs with very different runtimes. Other metrics of interest are makespan, i.e., the time to finish a given set of jobs, and Jain's fairness index [44] to measure how close the cluster scheduler comes to the desired allocations.

## 8.2 How does DagPS do in experiments?

### 8.2.1 *Job Completion Time*

Relative to `Tez`, Figure 10 shows that DagPS improves half of the DAGs by 19–31% across various benchmarks. One quarter of the DAGs improve by 30–49%. We see occasional

| Workload | `Tez+CP` | `Tez+Tetris` | DagPS |
|---|---|---|---|
| TPC-DS | +2.1% | +8.2% | +30.9% |
| TPC-H | +4.3% | +9.6% | +27.5% |

**Table 3: Makespan, gap from `Tez`.**

| Workload | Scheme | 2Q vs. 1Q Perf. Gap | Jain's fairness index 10s | 60s | 240s |
|---|---|---|---|---|---|
| TPC-DS | **Tez** | –13% | 0.82 | 0.86 | 0.88 |
| | **Tez+DRF** | –12% | 0.85 | 0.89 | 0.90 |
| | **Tez+Tetris** | –10% | 0.77 | 0.81 | 0.92 |
| | **DagPS** | +2% | 0.72 | 0.83 | 0.89 |

**Table 4: Fairness: Shows the performance gap and Jain's fairness index when used with 2 queues (even share) versus 1 queue. Here, a fairness score of 1 indicates perfect fairness.**

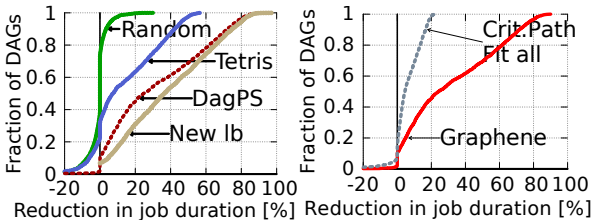
regressions. Up to 5% of the jobs slow down with DagPS; the maximum slowdown is 16%. We found this to be due to two reasons. (a) Noise from runtime artifacts such as stragglers and task failures and (b) Imprecise profiles: in all of our experiments, we use a single profile (the average) for all tasks in a stage but due to reasons such as data-skew, tasks in a stage can have different resource needs and durations. The table in Fig. 10 shows results for other benchmarks; we see that DAGs from E-Hive see the smallest improvement (19% at median) because the DAGs here are mostly two stage map-reduce jobs. The other benchmarks have more complex DAGs and hence receive sizable gains.

Relative to the alternatives, Figure 10 shows that DagPS is 15% to 34% better. `Tez + CP` achieves only marginal gains over `Tez`, hinting that critical path scheduling does not suffice. The exception is the BigBench dataset where about half the queries are dominated by work on the critical path. `Tez + Tetris` comes closest to DagPS because Tetris' packing logic reduces fragmentation. But, the gap is still substantial, since Tetris ignores dependencies. In fact, we see that `Tez + Tetris` does not consistently beat `Tez + CP`. Our takeaway is that considering both dependencies and packing can substantially improve DAG completion time.

Where do the gains come from? Figure 11 offers more detail on an example experimental run. DagPS keeps more tasks running on the cluster and hence finishes faster (Fig. 11a). The other schemes take over 20% longer. To run more tasks, DagPS gains by reducing fragmentation and by overbooking fungible resources. Comparing Fig. 11b with Figs. 11c–11d, the average allocation of all resources is higher with DagPS. Occasionally, DagPS allocates over 100% of the network and disk. `Tez + Tetris`, the closest alternative, has fewer tasks running at all times because (a) it does not overbook (all resource usages are below 100% in Fig. 11c) and (b) it ignores dependencies and packs greedily leading to a worse packing of the entire DAG. `Tez + CP` is impacted negatively by two effects: (a) ignoring disk and network usage leads to arbitrary over-allocation (the "total" resource usage is higher because, due to saturation, tasks hold on to allocations for longer) and (b) due to fragmentation, many fewer tasks run on average. Together these lead to low task throughput and job delays.

### 8.2.2 *Makespan*

To evaluate makespan, we make one change to experiment setup– all jobs arrive within the first few minutes. Everything




(a) DagPS vs. Baselines (b) DagPS vs. Alternatives

**Figure 12: Comparing DagPS with other schemes. We removed the lines from `CG` and `StripPart` because they hug $x = 0$; see Table 5.**

else remains the same. Table 3 shows the gap in makespan for different cases. Due to careful packing, DagPS sustains high cluster resource utilization, which in turn enables individual jobs to finish quickly: makespan improves 31% relative to `Tez` and over 20% relative to alternatives.

### 8.2.3 *Fairness*

Can we improve performance while also being fair? Intuitively, fairness may hurt performance since the task scheduling order needed for high performance (e.g., packability or dependencies) differs from the order that ensures fairness. To evaluate fairness, we make one change to the experiment set up. The jobs are evenly and randomly distributed among two queues and the scheduler has to divide resources evenly.

Table 4 reports the gap in performance (median job completion time) for each scheme when run with two queues vs. one queue. We see that `Tez`, `Tez + DRF` and `Tez + Tetris` lose over 10% in performance relative to their one queue counterparts. The table shows that with two queues, DagPS has a small gain (perhaps due to experimental noise). Hence, relatively, DagPS performs even better than the alternatives if given more queues (30% gap at one queue in Fig. 10a translates to a 40% gap at two queues). But why? Table 4 also shows Jain's fairness index computed over 10s, 60s and 240s time windows. We see that DagPS is less fair at short timescales but is indistinguishable at larger time windows. This is because DagPS is able to bound unfairness (§5); it leverages some short-term *slack* from precise fairness to make scheduling choices that improve performance.

## 8.3 Comparing with alternatives

We use simulations to compare a much wider set of best-of-breed algorithms (§8.1) on the much larger DAGs that ran in the production clusters. We mimic the actual dependencies, task durations and resource needs from the cluster.

Figure 12 compares the schedules constructed by DagPS with that from other algorithms. Table 5 reads out the gaps at various percentiles. We observe that DagPS's gains at the end of schedule construction are about the same as those obtained at runtime (Figure 10). This is interesting because the runtime component only softly enforces the desired schedules from all the jobs running simultaneously in the cluster. It appears that any loss in performance from not adhering to the desired schedule are made up by the gains from better packing and trading off some short-term unfairness.

Second, DagPS's gains are considerable compared to the alternatives. `CP` and `Tetris` are the closest. The reason is that DagPS looks at the entire DAG and places the troublesome tasks first, leading to a more compact schedule overall.

Third, when tasks have unit durations and nicely shaped demands, `CG` (Coffman-Graham [30]) is at most 2 times optimal. However, it does not perform well on production DAGs that have diverse demands for resources and varying durations. Some recent extensions to `CG` handle heterogeneity but ignore fragmentation issues when resources are divided across many machines [47].

Fourth, `StripPart` [20] is the best known algorithm that combines resource packing and task dependencies. It yields an $O(\log n)$-approx ratio on a DAG with $n$ tasks [20]. The key idea is to partition tasks into *levels* such that all dependencies go across levels. The primary drawback with `StripPart` is that it prevents overlapping independent tasks that happen to be in different levels. A secondary drawback is that the recommended packers (e.g., [60]) do not support multiple resources and vector packing. We see that in practice `StripPart` under-performs the simpler heuristics.

| | | $25^{th}$ | $50^{th}$ | $75^{th}$ | $90^{th}$ |
|---|---|---|---|---|---|
| **DagPS** | | 7 | 25 | 57 | 74 |
| Random | | −2 | 0 | 1 | 4 |
| Crit.Path | Fit cpu/mem | −2 | 0 | 2 | 1 |
| | Fit all | 1 | 4 | 13 | 16 |
| Tetris | Fit all | 0 | 7 | 29 | 42 |
| Strip Part. | Fit all | 0 | 1 | 12 | 27 |
| Coffman-Graham. | Fit all | 0 | 1 | 12 | 26 |
| | Fit cpu/mem | −2 | 0 | 0 | 2 |

**Table 5: Reading out the gaps from Figure 12; comparing DagPS vs. Alternatives. Each entry is the improvement relative to `BFS`.**

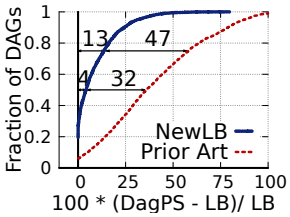


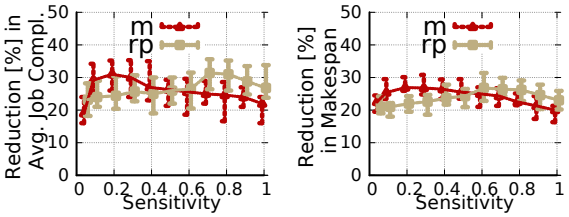

**Figure 13: Comparing DagPS with lower bounds.**

## 8.4 How close is DagPS to Optimal?

Figure 13 compares DagPS with `NewLB` and the best previous lower bound $\max(\texttt{CPLen}, \texttt{TWork})$ (see §6). Since the optimal schedule is no shorter than the lower bound, the figure shows that DagPS is optimal for about 40% of DAGs. For half (three quarters) of the DAGs, DagPS is within 4% (13%) of the new lower bound. A gap still remains: for the worst 10% of DAGs, DagPS takes 25% longer. Manually examining these DAGs shows that `NewLB` is loose for most of them. However, the figure also shows that the `NewLB` improves upon previous lower bounds by almost 30% for most of the DAGs. We conclude that while more work remains towards a good lower bound, `NewLB` suffices to argue that DagPS is close to optimal for most of the production DAGs.

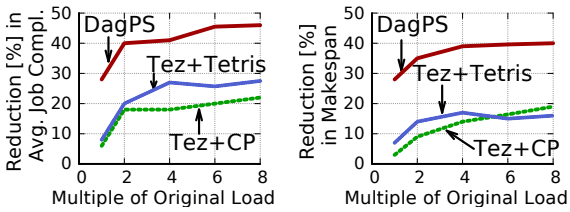
## 8.5 Sensitivity Analysis

We evaluate DagPS's sensitivity to parameter choices.




(a) Job Duration (b) Makespan

**Figure 14: DAGPS - sensitivity analysis.**




(a) Job Duration (b) Makespan

**Figure 15: DAGPS's gains increase with cluster load.**

**Packing vs. Shortest Remaining Processing Time (`srpt`):** Recall that we combine packing score and `srpt` using a weighted sum with $\eta$ (first box in Figure 5). Let $\eta$ be $m$ times the average over the two expressions that it combines. Here, we evaluate the sensitivity of the choice of $m$. Figure 14 shows the reduction in average job completion time (on left) and makespan (on right) for different values of $m$. Values of $m \in [0.1, 0.3]$ have the most gains. Lower values lead to worse average job completion time because the effect of `srpt` reduces. On the other hand, larger values lead to moderately worse makespan. Hence, we recommend $m = 0.2$.

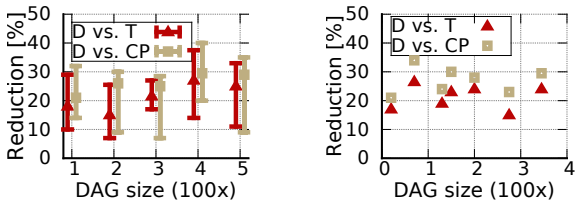
**Remote Penalty:** DagPS uses a remote penalty `rp` to prefer local placement. Our analysis shows that both job completion time and makespan improve the most when `rp` is between 15% and 30% (Fig. 14). Since `rp` is a multiplicative penalty, lower values of `rp` cause the scheduler to miss (non-local) scheduling opportunities whereas higher `rp` can over-use remote resources on the origin servers. We use `rp` = 0.8.

**Cluster Load:** We vary cluster load by reducing the number of available servers without changing the workload. Figure 15 shows the job completion times and makespan for a query set derived from TPC-DS. We see that both DagPS and the alternatives offer more gains at higher loads. This is to be expected since the need for careful scheduling and packing increases when resources are scarce. Gains due to DagPS increase by +10% at 2× load and by +15% at 6× load. However, the gap between DagPS and the alternatives remains similar across load levels.

## 9. APPLYING DagPS TO OTHER DOMAINS

We evaluate DagPS's effectiveness in scheduling the DAGs that arise in distributed compilation systems [4, 34] and request-response workflows for Internet services [45].

Distributed build systems speed up the compilation of large code bases [4, 34]. Each build is a DAG with dependencies between the various tasks (compilation, linking, test, code analysis). The tasks have different runtimes and have




(a) Distributed Build Systems: Compilation time (b) Request-response workflows: Query latency

**Figure 16: Comparing DAGPS (`D`) with Tetris (`T`) and Critical path scheduling (`CP`) on DAGs from two other domains.**

different resource profiles. Figure 16a shows that DagPS is 20% faster than `Tetris` and 30% faster than `CP` when scheduling the build DAGs from a production distributed build system. Each bar shows the median gain for DAGs of a certain size and the error bars are quartiles. The gains hold across DAG sizes/ types.

We also examine the DAGs that arise in datacenter-side request-response workflows for Internet-services [45]. For instance, a search query translates into a workflow of dependent RPCs at the datacenter (e.g., spell check before index lookup, video and image lookup in parallel). The RPCs use different resources, have different runtimes and often execute on the same server pool [45]. Over several workflows from a production service, Figure 16b shows that DagPS improves upon alternatives by about 24%.

These early results, though preliminary, are encouraging and demonstrate the generality of our work.

## 10. RELATED WORK

To structure the discussion, we ask four questions: (Q1) does a scheme consider both packing and dependencies, (Q2) does it make realistic assumptions, (Q3) is it practical to implement in cluster schedulers and, (Q4) does it consider multiple objectives such as fairness? To the best of our knowledge, DagPS is unique in positively answering these four questions.

`Q1 : NO`. Substantial prior work ignores dependencies but packs tasks with varying demands for multiple resources [26, 66, 37, 60, 73]. The best results are when the demand vectors are *small* [21]. Other work considers dependencies but assumes homogeneous demands [36, 30]. A recent multi-resource packing scheme, Tetris [37], succeeds on the three other questions but does not handle dependencies. Hence, we saw in §8 that it performs poorly when scheduling DAGs. Further, Tetris has poor worse-case performance (up to $2d$ times off, see Figure 19) and can be arbitrarily unfair.

`Q1 : YES, Q2 : NO`. The packing+dependencies problem has been considered at length under the keyword *job-shop scheduling* [48, 31, 35, 63]. Most results assume that jobs are known apriori (i.e., the offline case). See [47] for a survey. For the online case (the version considered here), no algorithms with bounded competitive ratios are known [52, 53]. Some other notable work assumes only two resources [22], applies only for a chain but not a general DAG [18] or assumes one

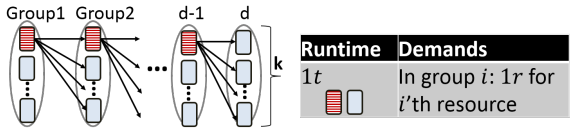



**Figure 17: A counter-example DAG that shows any scheduler not considering DAG structure will be $\Omega(d)$ times OPT.**

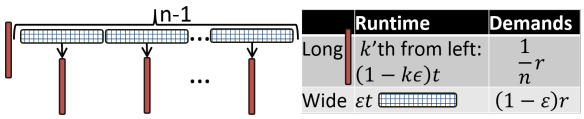



**Figure 18: An example DAG where critical path scheduling is $O(n)$ times OPT where $n$ is the number of nodes in the DAG.**

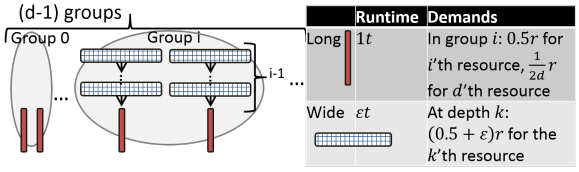



**Figure 19: An example DAG where Tetris [37] is $2d-2$ times OPT when tasks use $d$ kinds of resources.**

cluster-wide resource pool [49].

Q3 : NO. All of the schemes listed above consider one DAG at a time and are not easily adaptable to the online case when multiple DAGs share a cluster. Work on related problems such as VM allocation [29] also considers multi-resource packing. However, cluster schedulers have to support roughly two to three orders of magnitude higher rate of allocation (tasks are more numerous than VMs).

Q3 : YES, Q1 : NO. Several notable works in cluster scheduling exist such as Quincy [43], Omega [62], Borg [69], Kubernetes [9] and Autopilot [41]. None of these combine multi-resource packing with DAG-awareness. Many do neither. Job managers such as Tez [3] and Dryad [42] use simple heuristics such as breadth-first scheduling which perform quite poorly in our experiments.

Q4 : NO. There has been much recent work on novel fairness schemes to incorporate multiple resources [33] and be work-conserving [27]. Several applications arise especially in scheduling coflows [28, 54]. We note that these fairness schemes neither pack nor are DAG-aware. DagPS can incorporate these fairness methods as one of the multiple objectives and trades off bounded unfairness for performance.

## 11. CONCLUDING REMARKS

DAGs are indeed a common scheduling abstraction. However, we found that existing algorithms make several key assumptions that do not hold in practical settings. Our solution, DagPS is an efficient online solution that scales to large clusters. We experimentally validated that it substantially improves the scheduling of DAGs in both synthetic and emulated production traces. The core contributions are threefold: (1) constructing a good schedule by placing tasks out-of-order on to a virtual resource-time space, (2) an online heuristic that softly enforces the desired schedules and helps with other concerns such as packing and fairness, and (3) an improved lower bound that lets us show that our heuristics are close to optimal. Much of these innovations use the fact that job DAGs consist of groups of tasks (in each stage) that have similar durations, resource needs and dependencies. We intend to contribute our DagPS implementation to Apache YARN/Tez projects. As future work, we are considering applying these DAG scheduling ideas to related domains, most notably scheduling the coflows with dependencies that arise in geo-distributed analytics [39, 58, 70, 40].

## A. VALUE OF DAG AWARENESS

*Proof of Lemma 1:* Figure 17 shows an adversarial DAG for which *any* scheduler that ignores dependencies will take $\Omega(d)$ times OPT. Assume cluster capacity is $1r$ for each of the $d$ resources. The DAG has $d$ groups, each having a task filled with red dashes that is the parent of all the tasks in the next group. This information is unavailable (and unused) by schedulers that do not consider the DAG structure. Hence, regardless of which order the scheduler picks tasks, an adversary can choose the last task in a group to be the red one. Hence, such schedulers will take $kdt$ time. OPT only requires $(k+d-1)t$ since it can schedule the red tasks first (in $(d-1)t$) and afterwards one task from each group can run simultaneously ($kt$ more steps). We use Yao's max-min principle [55] (the lower bound on any randomized algorithm is the same as lower bound on deterministic algorithms with randomized input) to extend the counter-example. If we randomize the choice of the red task, the expected time at which the red task will finish is $kt/2$ and hence the expected schedule time is $k(d+1)t/2$ which is still $\Omega(d)$ times OPT. □

## B. WORST-CASE DAG EXAMPLES

*Proof of Lemma 2:* Figure 18 shows an example DAG where CPSched takes $n$ times worse than OPT for a DAG with $2n$ tasks. The critical path lengths of the various tasks are such that CPSched alternates between one long task and one wide task left to right. However, it is possible to overlap all of the long running tasks. This DAG completes at $\sim nt$ and $\sim 1t$ with CPSched and OPT respectively. Fig. 19 shows an example where Tetris [37] is $2d-2$ times OPT. As in the above example, all long tasks can run together, hence OPT finishes in $1t$. Tetris greedily schedules the task with the highest dot-product between task demands and available resources. The DAG is constructed such that whenever a long task is runnable, it will have a higher score than any wide task. Further, for every long task that is not yet scheduled, there exists at least one wide parent that cannot overlap any long task that may be scheduled earlier. Hence, Tetris takes $(2d-2)t$ which is $(2d-2)$ times OPT. Combining these two principles, we conjecture that it is possible to find similar examples for any scheduler that ignores dependencies or is not resource-aware. □

## 12. REFERENCES

[1] 43 bigdata platforms and bigdata analytics so□ware. http://bit.ly/1DROqgt.

[2] Apache Hadoop. http://hadoop.apache.org.

[3] Apache Tez. http://tez.apache.org/.

[4] Bazel. http://bazel.io/.

[5] Big-Data-Benchmark. http://bit.ly/1HlFRH0.

[6] Condor. http://research.cs.wisc.edu/htcondor/.

[7] Hadoop: Fair scheduler/ slot fairness. http://bit.ly/1PfsT7F.

[8] Hadoop YARN Project. http://bit.ly/1iS8xvP.

[9] Kubernetes. http://kubernetes.io/.

[10] Market research on big-data-as-service o□erings. http://bit.ly/1V6TXV4.

[11] TPC-DS Benchmark. http://bit.ly/1J6uDap.

[12] TPC-H Benchmark. http://bit.ly/1KRK5gl.

[13] Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society* (1998).

[14] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[15] Agarwal, S., Kandula, S., Burno, N., Wu, M.-C., Stoica, I., and Zhou, J. Re-optimizing data parallel computing. In *NSDI* (2012).

[16] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *SIGCOMM* (2008).

[17] Ananthanarayanan, G., et al. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI* (2010).

[18] Anderson, E., Beyer, D., Chaudhuri, K., Kelly, T., Salazar, N., Santos, C., Swaminathan, R., Tarjan, R., Wiener, J., and Zhou, Y. Value-maximizing deadline scheduling and its application to animation rendering. In *SPAA* (2005).

[19] Armbrust, M., et al. Spark sql: Relational data processing in spark. In *SIGMOD* (2015).

[20] Augustine, J., Banerjee, S., and Irani, S. Strip packing with precedence constraints and strip packing with release times. In *SPAA* (2006), ACM.

[21] Azar, Y., Cohen, I. R., Fiat, A., and Roytman, A. Packing small vectors. In *SODA* (2016).

[22] Belkhale, K. P., and Banerjee, P. An approximate algorithm for the partitionable independent task scheduling problem. *Urbana 51* (1990), 61801.

[23] Bruno, N., Jain, S., and Zhou, J. Continuous cloud-scale query optimization and processing. *VLDB* (2013).

[24] Chaiken, R., et al. SCOPE: Easy and E□cient Parallel Processing of Massive Datasets. In *VLDB* (2008).

[25] Chambers, C., et al. Flumejava: easy, e□cient data-parallel pipelines. In *PLDI* (2010).

[26] Chandra, C., and Sanjeev, K. On multidimensional packing problems. *SIAM J. Comput.* (2004).

[27] Chowdhury, M., Liu, Z., Ghodsi, A., and Stoica, I. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI* (2016).

[28] Chowdhury, M., Zhong, Y., and Stoica, I. E□cient co□ow scheduling with varys. In *SIGCOMM* (2014).

[29] Chowdhury, N., Rahman, M., and Boutaba, R. Virtual Network Embedding with Coordinated Node and Link Mapping. In *INFOCOM* (2009).

[30] Coffman, E.G., J., and Graham, R. Optimal scheduling for two-processor systems. *Acta Informatica* (1972).

[31] Czumaj, A., and Scheideler, C. A new algorithm approach to the general lovÁsz local lemma with applications to scheduling and satis□ability problems (extended abstract). In *STOC* (2000).

[32] Dean, J., and Ghemawat, S. Mapreduce: Simpli□ed data processing on large clusters. In *OSDI* (2004).

[33] Ghodsi, A., et al. Dominant Resource Fairness: Fair Allocation Of Multiple Resource Types. In *NSDI* (2011).

[34] Gliboric, M., Schulte, W., Prasad, C., van Velzen, D., Narsamdya, I., and Livshits, B. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *OOPSLA* (2014).

[35] Goldberg, L. A., Paterson, M., Srinivasan, A., and Sweedyk, E. Better approximation guarantees for job-shop scheduling. In *SODA* (1997).

[36] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics* (1969).

[37] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource Packing for Cluster Schedulers. In *SIGCOMM* (2014).

[38] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: a platform for □ne-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 22–22.

[39] Hung, C.-C., Golubchik, L., and Yu, M. Scheduling jobs across geo-distributed datacenters. In *SOCC* (2015).

[40] Hung, C.-C., Golubchik, L., and Yu, M. Scheduling jobs across geo-distributed datacenters. In *SOCC* (2015).

[41] Isard, M. Autopilot: Automatic Data Center Management. *OSR* (2007).

[42] Isard, M., et al. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Eurosys* (2007).

[43] Isard, M., et al. Quincy: Fair Scheduling For Distributed Computing Clusters. In *SOSP* (2009).

[44] Jain, R., Chiu, D., and Hawe, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR cs.NI/9809099* (1998).

[45] Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., and Yan, C. Speeding up distributed request-response work□ows. In *SIGCOMM* (2013).

[46] Ke, Q., Isard, M., and Yu, Y. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *Eurosys* (2013).

[47] Kwok, Y., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)* (1999).

[48] Leighton, F. T., Maggs, B. M., and Rao, S. Universal packet routing algorithms. In *FOCS* (1988).

[49] Lepère, R., Trystram, D., and Woeginger, G. J. Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints. *International Journal of Foundations of Computer Science* (2002).

[50] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI* (2008).

[51] Maggs, B. M., auf der Heide, F. M., Voecking, B., and Westermann, M. Exploiting locality for data management in systems of limited bandwidth. In *FOCS* (1997).

[52] Mastrolilli, M., and Svensson, O. (acyclic) job shops are hard to approximate. In *FOCS* (2008).

[53] Monaldo, M., and Ola, S. Improved bounds for □ow shop scheduling. In *ICALP* (2009).

[54] Mosharaf Chowdhury, I. S. E□cient co□ow scheduling without prior knowledge. In *SIGCOMM* (2015).

[55] Motwani, R., and Raghavan, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[56] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: Distributed, low latency scheduling. In *SOSP* (2013).

[57] Panigrahy, R., et al. Heuristics for Vector Bin Packing. In *MSR TR* (2011).

[58] Pu, Q., Ananthanarayanan, G., Bodik, P., Kandula, S., Akella, A., Bahl, P., and Stoica, I. Low latency geo-distributed analytics. In *SIGCOMM* (2015).

[59] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[60] Schiermeyer, I. Reverse-□t: A2-optimal algorithm for packing rectangles. In *Proceedings of the Second Annual European Symposium on Algorithms* (1994).

[61] Schurman, E., and Brutlag, J. □e User and Business Impact of Server Delays, Additional Bytes, and Http Chunking in Web Search. http://velocityconf.com/velocity2009/public/schedule/detail/8523, 2009.

[62] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: Flexible, scalable schedulers for large compute clusters. In *EuroSys* (2013).

[63] Shmoys, D. B., Stein, C., and Wein, J. Improved approximation algorithms for shop scheduling problems. *SIAM J. Comput.* (1994).

[64] Shreedhar, M., et al. E□cient fair queueing using de□cit round robin. In *SIGCOMM* (1995).

[65] Singh, A., et al. Jupiter rising: A decade of clos topologies and centralized control in googleâĂŹs datacenter network. In *SIGCOMM* (2016).

[66] Sungjin, I., Nathaniel, K., Janardhan, K., and Debmalya, P. Tight bounds for online vector scheduling. In *FOCS* (2015).

[67] Suresh, L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *NSDI* (2015).

[68] Thusoo, A., et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB* (2009).

[69] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *EuroSys* (2015).

[70] Vulimiri, A., Curino, C., Godfrey, P. B., Jungblut, T., Padhye, J., and Varghese, G. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI* (2015).

[71] Woeginger, G. J. □ere Is No Asymptotic PTAS For Two-Dimensional Vector Packing. In *Information Processing Letters* (1997).

[72] Yossi, A., Ilan, C., Seny, K., and Bruce, S. Tight bounds for online vector bin packing. In *STOC* (2013).

[73] Yossi, A., Ilan Reuven, C., and Iftah, G. □e loss of serving in the dark. In *STOC* (2013).

[74] Zaharia, M., et al. Spark: Cluster computing with working sets. Tech. Rep. UCB/EECS-2010-53, EECS Department, University of California, Berkeley, 2010.