Altruistic Scheduling in Multi-Resource Clusters

Robert Grandl, *University of Wisconsin—Madison;* Mosharaf Chowdhury, *University of Michigan;* Aditya Akella, *University of Wisconsin—Madison;* Ganesh Ananthanarayanan, *Microsoft*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl_altruistic

ISBN 978-1-931971-33-1

Altruistic Scheduling in Multi-Resource Clusters

Robert Grandl[1], Mosharaf Chowdhury[2], Aditya Akella[1], Ganesh Ananthanarayanan[3]
[1] *University of Wisconsin-Madison* [2]*University of Michigan* [3]*Microsoft*

Abstract

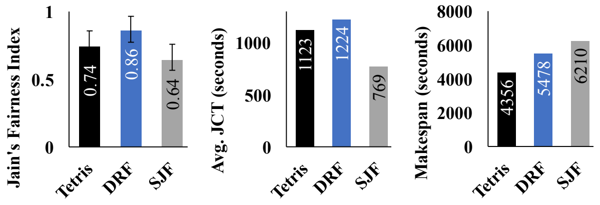
Given the well-known tradeoffs between fairness, performance, and efficiency, modern cluster schedulers often prefer instantaneous fairness as their primary objective to ensure performance isolation between users and groups. However, instantaneous, short-term convergence to fairness often does not result in noticeable long-term benefits. Instead, we propose an *altruistic*, long-term approach, CARBYNE, where jobs yield fractions of their allocated resources without impacting their own completion times. We show that leftover resources collected via altruisms of many jobs can then be rescheduled to further secondary goals such as application-level performance and cluster efficiency without impacting performance isolation. Deployments and large-scale simulations show that CARBYNE closely approximates the state-of-the-art solutions (e.g., DRF [27]) in terms of performance isolation, while providing 1.26× better efficiency and 1.59× lower average job completion time.

1 Introduction

Resource scheduling remains a key building block of modern data-intensive clusters. As data volumes increase and the need for analysis through multiple lenses explode [1, 2, 12, 23, 34, 41, 43, 45, 55], diverse coexisting jobs from many users and applications contend for the same pool of resources in shared clusters.

Consequently, today's cluster schedulers [9, 17, 33, 51] have to deal with multiple resources [14, 20, 27, 29, 38], consider jobs with complex directed acyclic graph (DAG) structures [19, 29, 55], and allow job-specific constraints [15, 28, 35, 54, 56]. Schedulers must provide performance isolation between different users and groups through fair resource sharing [9, 16, 17, 27, 29, 36, 56], while ensuring performance (low job completion times) and efficiency (high cluster resource utilization).

However, simultaneously optimizing fairness, performance, and efficiency is difficult. Figure 1 demonstrates the tradeoff space by comparing three schedulers – dominant resource fairness (DRF) [27] for multi-resource fairness, shortest-job-first (SJF) [26] for minimizing the average job completion time (JCT), and Tetris [29] for increasing resource utilization – implemented in Apache YARN [51]: each scheduler outperforms its counterparts only in a preferred metric and *significantly* underperforms in the secondary metrics. In practice, many production schedulers [9, 10, 17, 33, 48, 51, 52] settle for performance isolation as their primary objective and focus on quick convergence to *instantaneous* fair allocations, while considering performance and efficiency as *best-effort*, secondary goals, often without an explicit focus on them.



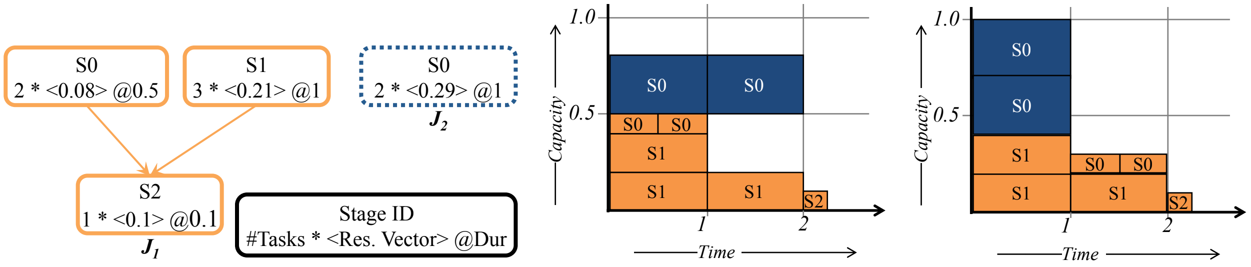
(a) Inter-job fairness **(b)** Job performance **(c)** Cluster efficiency

Figure 1: DRF [27], SJF [26], and Tetris [29] on a TPC-DS [7] workload on a 100-machine cluster. All jobs arrived together. Higher is better in (a),[1] while the opposite is true in (b) and (c).

In this paper, we revisit the three-way tradeoff space: *is it possible to ensure performance isolation (i.e., fairness) and still be competitive with the best approaches for the secondary metrics (job completion time and cluster utilization)?* We highlight two key characteristics. First, distributed data-parallel jobs share an *all-or-nothing* characteristic [14, 15, 21, 56]: a parallel job cannot complete until all its tasks have completed. Because of this, aggressively using all of the fair share often does not translate into noticeable benefits in terms of secondary metrics. In fact, it can hurt the average JCT and cluster efficiency. Second, users only observe the outcome of performance isolation when their jobs complete, and they care less for instantaneous fair-share guarantees.

Thus, we propose an *altruistic*, *long-term* approach based on a simple insight: jobs yield fractions of their currently allocated resources – referred to as *leftover* resources – to be redistributed to further secondary objectives. As long as jobs can contribute resources and still complete without taking additional time, the effects of such redistribution remain transparent to users. (We

[1] In Figure 1a, we calculated the average of Jain's fairness index [37] over 60-second intervals; error bars represent the minimum and the maximum values.



(a) Two DAGs w/ their dependencies and requirements **(b)** Aggressive scheduling **(c)** Altruistic scheduling

Figure 2: Opportunities in altruistic scheduling for two DAGs contending on a single bottleneck resource. Job J_1 (orange/light/solid line) has three stages and Job J_2 (blue/dark/dotted line) has one stage (a), where the number of tasks in each stage along with their expected resource requirements and durations are shown as specified in the legend. Assuming each task can start and complete its execution in the specified duration given its required resources, (b)–(c) depict the allocations of the bottleneck resource (vertical axis) for different approaches: The average completion time for (b) traditional schedulers (i.e., single- or multi-resource fairness across jobs and efficient packing within each job) is 2.05 time units; and (c) CARBYNE with altruistic scheduling is 1.55 time units.

prove that an altruistic scheduler can guarantee this in the offline setting).

Indeed, jobs have ample opportunities for altruisms in production (§2). For example, 50% of the time, at least 20% of the allocated resources can be used as leftover (i.e., yielded altruistically) at Microsoft's production analytics clusters. Given a fixed amount of resources, a job may be altruistic whenever it cannot simultaneously schedule all of its runnable tasks – i.e., when it can choose which tasks to schedule at that instant and which ones to schedule later. As clusters are shared between more users and DAG complexity (e.g., the number of barriers, total resource requirements, and critical path length [39]) increases, the amount of leftover resources increases too.

Leftover resources collected via altruisms of many jobs introduce an *additional degree of freedom* in cluster scheduling. We leverage this flexibility in CARBYNE that decides how to determine and redistribute the leftover, so as to improve one or both of performance and efficiency metrics as the cluster operator sees fit, while providing the same level of user-observable performance isolation as the state-of-the-art (§3). Specifically, given a share of the cluster resources (computed by a fair-sharing scheme like DRF), CARBYNE's intra-job scheduler computes task schedules further into the future (since it knows the DAG of the job upfront). This allows each job to compute, at any given time, which runnable tasks to schedule and how much of its fair share to contribute altruistically. CARBYNE's inter-job scheduler uses the resulting leftover resources to: (a) preferentially schedule tasks from jobs that are closest to completion; then (b) pack remaining tasks to maximize efficiency. The order of (a) and (b) can be reversed to prioritize efficiency over JCTs. Note that our focus is not on improving the scheduling heuristics for the individual metrics themselves, but rather on combining them with altruism. To that end, our solution can work with any of the scheduling heuristics designed for fairness (say, capacity scheduler [5]), completion time, and cluster efficiency (say, DAGPS [30, 31]).

Prior work have also attempted to simultaneously meet multiple objectives [18, 21, 26, 29–31, 38, 47]. The main difference is that given some resources (e.g., according to some fair allocation) prior approaches (notably [29–31]) adopt eager scheduling of tasks, whereas CARBYNE schedules altruistically by delaying tasks in time as much as possible to accommodate those that are in greater need of running now. As our examples in Section 2 and results in Section 5 show, such an altruism-based approach offers better control over meeting global objectives. In particular, compared to [29–31], CARBYNE is better at meeting fairness guarantees, while offering similar performance along other objectives.

We have implemented CARBYNE as a pluggable scheduler on Apache YARN [51] (§4). Any framework that runs on YARN can take advantage of CARBYNE. We deployed and evaluated CARBYNE on 100 bare-metal machines using TPC-DS [7], TPC-H [8], and BigBench [4] queries and also by replaying production traces from Microsoft and Facebook in trace-driven simulations (§5). In deployments, CARBYNE provides 1.26× better efficiency and 1.59× lower average completion time than DRF, while closely approximating it in fairness. In fact, CARBYNE's performance and efficiency characteristics are close to that of SJF and Tetris, respectively. Only 4% jobs suffer more than 0.8× slowdown (with the maximum slowdown being 0.62×) as CARBYNE temporally reorganizes tasks to improve all three metrics. We observed slightly higher benefits in simulations for complex DAGs as well as for simple MapReduce jobs. Furthermore, we found that CARBYNE performs well even in the presence of misestimations of task demands (needed for packing) and even when some jobs are not altruistic.

	# Stages		# Barriers		Input Work	
Workload	50th	95th	50th	95th	50th	95th
Microsoft	13	121	4	13	34%	85%
TPC-DS	8	23	1	4	11%	88%
TPC-H	8	12	2	4	46%	82%
BigBench	7	19	2	6	24%	70%

Table 1: Structural diversity in various workloads. Each group of columns reads out percentile distributions.

	CP Length		# Disjoint Paths	
Workload	50th	95th	50th	95th
Microsoft	7	17	6	34
TPC-DS	5	8	4	15
TPC-H	5	7	3	7
BigBench	5	8	2	10

Table 2: Diversity in length of the critical path (CP) and the number of disjoint paths in DAGs across various workloads.

2 Motivation

This section demonstrates benefits of altruistic scheduling using an example (§2.1) and quantitatively analyzes altruism opportunities (§2.2) in DAG workloads.

2.1 Illustration of Altruism

Modern schedulers focus on *instantaneous* (i.e., short-term) optimizations and take greedy decisions. We hypothesize that the focus on the short-term restricts their flexibility in optimizing secondary objectives. Instead, if we relax them via *short-term altruisms* and focus on long-term optimizations, we can enable disproportionately larger flexibility during scheduling, which, in turn, can translate into significant improvements.

Consider Figure 2 that compares two schedules assuming both jobs arrive at the same time. Existing schedulers (Figure 2b) perform independent scheduling both across and within jobs, resulting in an average completion time of 2.05 time units (by allocating equal share of resources to the jobs). This holds for *any* combination of today's schedulers – max-min [36] or dominant resource fairness (DRF) [27] across different jobs and breadth-first order [2, 55], critical path method (CPM) [39, 40], or packing [29] within each job – because they are independent by design to optimize for short-term objectives.

In contrast, Carbyne takes a long-term approach (Figure 2c), where the intra-job scheduler of J_1 altruistically gives up some of its resources (since S2 in J_1 is a barrier), which can better serve J_2. As long as tasks in stages $S0$ and $S1$ finish by 2 time units, J_1's completion time will not change. By ignoring short-term greed, J_1 can improve J_2's completion time, resulting in an average completion time of 1.55 time units (1.3× better). [2]

Note that this example considers only one resource, only two jobs, and simplistic DAGs. As complexity increases across all these dimensions, Carbyne can be even more effective (§5).

2.2 Opportunities in Analytics DAGs

In this section, we analyze the potential for altruism. We analyzed query DAGs from a large Microsoft production trace, TPC-DS [7], TPC-H [8], and BigBench [4] benchmarks. We first quantify the amount of leftover resources for altruism and then correlate it with relevant DAG properties. For a detailed analysis of all DAG properties in production, we refer interested readers to [30, 31].

How much resources can be considered as leftover and used for altruistic scheduling? To answer this, we computed the fraction of allocated compute, memory, disk in/out, and network in/out bandwidths that could have been reallocated without affecting any job's completion time at Microsoft for each scheduling event such as job and task arrivals and completions. We found that across thousands of scheduling events, 50% of the time, at least 20% of the allocated resources – $\langle 35\%, 43\%, 24\%, 21\%, 28\%, 24\% \rangle$ for the resources listed above – could be used as leftover and rescheduled via altruistic scheduling.

We next analyze the properties of the jobs that matter the most for altruism: total stages, number of stages with multiple parents (i.e., barriers), length of the DAG's critical path, and the number of disjoint paths in the DAG.

Stage-Level Observations The number of stages in a DAG provides an approximation of its complexity. In our traces, a DAG has at least 7 stages at median and up to 23 at the 95th percentile (Table 1). These numbers are significantly higher for production traces.

Recall that Carbyne is altruistic without hampering JCT. A key factor dictating this is the number of barriers, i.e., stages whose children must wait until they finish. Examples include aggregation and join stages, or even final output stages. We observed the presence of multiple barriers in most DAGs (Table 1); also quantified in [15].

Path-Level Observations Next, we considered the length of the critical path of the DAG [39] as well as the number of disjoint paths. We defined each sequence of stages that can run in parallel with other sequences of different stages as a disjoint path. Table 2 shows that the median length of the critical path of DAGs is 5 in these workloads, and many DAGs have multiple disjoint paths, further enabling altruistic scheduling; [30, 31] has a detailed description and quantification of DAG critical paths.

[2] For this example, packing across job boundaries results in an average JCT of 1.55 time units. However, as shown in Figure 1b, packing does not always lead to the best average JCT, and it is not difficult to construct a similar example.

Correlating DAG properties and leftover resources To understand which of these DAG properties have the highest impact, we calculated correlations over time between the amount of leftover resources via CARBYNE and variabilities in each of the aforementioned attributes. We found that the number of barriers in a DAG has the highest positive correlation (0.75). The more barriers a DAG contains, the more opportunities it has to contribute. For example, in Figure 2c, the barrier $S2$ of DAG J_1 enabled its altruism. The number of stages, critical path length, and the number of disjoint paths also have high positive correlations of 0.66, 0.71 and 0.57, respectively.

3 Altruistic Multi-Resource Scheduling

In this section, we present an online altruistic multi-resource DAG scheduler. First, we define the problem along with our assumptions (§3.1). Next, we discuss desirable properties of an ideal DAG scheduler and associated tradeoffs (§3.2). Based on our understanding, we develop an offline altruistic scheduler in two steps (§3.3): determining how much a job can contribute to leftover and deciding how to distribute the leftover or yielded resources to other jobs. Finally, we analyze why the offline solution works well in the online scenario as well (§3.4).

3.1 Problem Statement

Given a collection of jobs – along with information about individual tasks' expected multi-resource requirements, durations, and DAG dependencies – we must schedule them such that each job receives a fair share of cluster resources, jobs complete as fast as possible, and the schedule is work-conserving. All information about individual jobs are unknown prior to their arrival.

3.2 Complexity and Desirable Properties

Offline DAG scheduling is NP-complete [25] for all the objectives[3] – fairness, performance, and efficiency – even when the entire DAG and completion times of each of its stages are known. In fact, polynomial-time optimal DAG scheduling algorithms exist for only three simple cases [22, 42], none of which are applicable to DAG scheduling in multi-resource clusters.

Because achieving optimality in all of the aforementioned objectives is impossible due to their tradeoffs [18, 21, 26, 29, 38, 47], we want to build a scheduler that improves performance and efficiency without sacrificing performance isolation. In the online case, we expect an ideal such scheduler to satisfy the following goals:

- *Fast completion:* Each DAG should complete as fast as possible.
- *Work conservation:* Available resources should not remain unused.
- *Starvation freedom:* No DAG should starve for arbitrarily long periods.[4]

3.3 Offline Altruistic Scheduling

Cluster schedulers typically operate in two levels: inter- and intra-job; i.e., between jobs and between tasks of each job. However, we want to consider three distinct scheduling components – the two above, and leftover scheduling. To this end, we first identify an *intermediate* degree of freedom in scheduling, and we discuss how to leverage that.

3.3.1 Solution Approach

Consider the offline problem of scheduling $|\mathbb{J}|$ DAG jobs ($\mathbb{J} = \{J_1, J_2, \ldots, J_{|\mathbb{J}|}\}$) that arrived at time 0. We start by observing that given a fixed share ($\overrightarrow{A_k} = \langle a_k^1, a_k^2, \ldots a_k^{|\overrightarrow{R}|}\rangle$) of $|\overrightarrow{R}|$ resources ($\overrightarrow{R} = \langle R^1, R^2, \ldots, R^{|\overrightarrow{R}|}\rangle$) by an inter-job scheduler, a DAG will take a minimum of amount of time (T_k for job J_k) to complete all its tasks, given their dependencies. However, as long as its allocation does not decrease – true in the offline case – it can decide *not to be aggressive* in using resources and still complete by T_k. Formally, we prove the following:

Theorem 3.1 *Altruism will not inflate any job's completion time in the offline case – i.e., unless new jobs arrive or existing jobs depart – for any inter-job scheduler.*

The proof follows from the fact that none of the T_k's invariants – namely, resource requirements, DAG structure, and allocation of J_k – change in the offline case.

We refer to resources that are not immediately required as *leftover* resources ($\overrightarrow{L_k} = \langle l_k^1, l_k^2, \ldots l_k^{|\overrightarrow{R}|}\rangle$), and we consider contributing to $\overrightarrow{L_k}$ to be an *altruistic* action. For example, in Figure 2c, J_1 can potentially have 0.29 units of leftover resources at time 0. Assuming its fixed resource share of 0.5 units, it would still be able to complete in 2.1 time units by delaying one more task from stage $S1$ to start at time 1. Note that J_1 is instead running that task at time 0 to ensure work conservation.

Given that multiple jobs can contribute leftover resources (§2.2), if we combine ($\overrightarrow{\mathbb{L}} = \sum_k \overrightarrow{L_k}$) and redistribute them across all running jobs, we can create new opportunities for improving the secondary metrics. We can now reformulate cluster scheduling as the following three questions:

1. *how to perform inter-job scheduling to maximize the amount of leftover resources?*
2. *how should an intra-job scheduler determine how much a job should contribute to leftover?* and
3. *how to redistribute the leftover across jobs?*

[3] Whether multi-resource fair DAG scheduling is NP-complete is unknown. DRF [27] results were proven for jobs where *all* tasks have the same resource requirements, which is not the case in multi-stage DAGs.

[4] We do not aim for stronger goals such as *guaranteeing bounded starvation*, because providing guarantees require resource reservation along with admission control. Even DRF does not provide any guarantees in the online case.

By addressing each one at each scheduling granularity – inter-job, intra-job, and leftover – we design an altruistic DAG scheduler (Pseudocode 1) that can compete with the best and outperform the rest in all three metrics (§5).

CARBYNE invokes Pseudocode 1 on job arrival, job completion, as well as task completion events. For each event, it goes through three scheduling steps to determine the tasks that must be scheduled at that point in time. While we use DRF [27], SRTF, and Tetris [29] for concrete exposition, CARBYNE can also equivalently use other schedulers such as DAGPS [30, 31] (instead of Tetris) for packing and capacity scheduler [5] (instead of DRF) for fairness.

3.3.2 Increasing Leftover via Inter-Job Scheduling

Modern clusters are shared between many groups and users [9, 10, 16, 17, 33, 48, 51, 52]. Consequently, the primary goal in most production clusters is performance isolation through slot-based [9, 10] or multi-resource [16, 27] fairness. We use a closed-form version of DRF [46] for inter-job scheduling in CARBYNE. Interestingly, because a single- or multi-resource fair scheduler enforces resource sharing between *all* running jobs, it elongates individual job completion times the most (compared to schedulers focused on other primary objectives). Consequently, fair schedulers provide the most opportunities for altruistic scheduling.

3.3.3 Determining Leftover for Individual Jobs

Given some resource share $\overrightarrow{A_k}$, CARBYNE's intra-job scheduler aims to maximize the amount of leftover resources from each job. We schedule only those tasks that *must* start running for J_k to complete within the next T_k duration, and we altruistically donate the rest of the resources for redistribution. Computing this is simple: we simply perform a reverse/backward packing of tasks from T_k to current time, packing tasks in time as close to T_k as possible, potentially leaving more resources available at earlier times (lines 17–21 in Pseudocode 1). For example, in Figure 2, stage $S2$ of job J_1 can only start after 2 time units. Hence, CARBYNE can postpone both tasks from $S0$ and one task from $S1$ until at least the first time unit, donating 0.29 units to leftover resources (0.08 for $S0$'s tasks and 0.21 for $S1$'s task). Similarly, job J_2 donates 0.21 units to leftover resources (its fair share of 0.5 less the resource used by one task in its $S0$ of 0.29), making it a sum of 0.5 leftover resource units.

Reverse packing uses the same principle as the intra-coflow scheduler used in Varys [21], where most flows in a coflow are slowed down so that all of them finish together with the longest running one. However, CARBYNE considers multiple resource types as well as dependencies, and unlike Varys, it does not hog CPU and memory.

Pseudocode 1 Altruistic DAG Scheduling

```
1:  procedure SCHEDULE(Jobs $\mathbb{J}$, Resources $\overrightarrow{R}$)
2:      $\{\overrightarrow{A_k}\}$, $\overrightarrow{\mathbb{L}}$ = INTERJOBSCHEDULER($\mathbb{J}$, $\overrightarrow{R}$)
3:      for all $J_k \in \mathbb{J}$ do
4:          $\overrightarrow{\mathbb{L}}$ += INTRAJOBSCHEDULER($J_k$, $\overrightarrow{A_k}$)
5:      end for
6:      LEFTOVERSCHEDULER($\mathbb{J}$, $\overrightarrow{\mathbb{L}}$)
7:  end procedure

8:  procedure INTERJOBSCHEDULER(Jobs $\mathbb{J}$, Resources $\overrightarrow{R}$)
9:      $\overrightarrow{L_{\text{DRF}}} = \overrightarrow{R}$      ▷ $\overrightarrow{L_{\text{DRF}}}$ tracks total unalloc. resources
10:     Calculate $\overrightarrow{A_k}$ using closed-form DRF [46] for all $J_k$
11:     for all $J_k \in \mathbb{J}$ do
12:         $\overrightarrow{L_{\text{DRF}}}$ -= $\overrightarrow{A_k}$
13:     end for
14:     return $\{A_k\}$ and $\overrightarrow{L_{\text{DRF}}}$
15: end procedure

16: procedure INTRAJOBSCHEDULER(Job $J_k$, Alloc. $\overrightarrow{A_k}$)
17:     Given $\overrightarrow{A_k}$, calculate $T_k$ using Tetris [29]
18:     Reverse parent-child task dependencies in $J_k$
19:     Calc. task start times Rev$(t_k^j)$ in the reversed DAG
20:     Actual$(t_k^j)$ = Rev$(t_k^j) + T_k -$ Dur$(t_k^j)$
21:     $\mathbb{T}_k = \left\{t_k^j : \text{Actual}(t_k^j) == 0\right\}$ ▷ Tasks that must start
22:     Schedule $t_k^j$; $\forall t_k^j \in \mathbb{T}_k$
23:     $\overrightarrow{L_k} = \overrightarrow{A_k} - \sum \text{Req}(t_k^j);\ t_k^j \in \mathbb{T}_k$
24:     return $\overrightarrow{L_k}$
25: end procedure

26: procedure LEFTOVERSCHEDULER(Jobs $\mathbb{J}$, Leftover $\overrightarrow{\mathbb{L}}$)
27:     $\mathbb{J}'$ = SORT_ASC ($\mathbb{J}$) in the SRTF order
28:     for all $J_k \in \mathbb{J}'$ do
29:         Schedule runnable tasks to maximize $\overrightarrow{\mathbb{L}}$ usage
30:     end for
31: end procedure
```

A sophisticated packer (like DAGPS [30, 31]) can better increase leftover resources. As opposed to our reverse packer that only considers the parents of currently runnable tasks, it could consider the entire DAG and its dependencies. It could also account for the nature of dependencies between stages (e.g., many-to-one, all-to-all). The more CARBYNE can postpone tasks into the future without affecting a DAG's completion time, the more leftover resources it has for altruistic scheduling.

3.3.4 Redistribution via Leftover Scheduling

Given the pool of leftover resources $\overrightarrow{\mathbb{L}}$ from all jobs (0.5 units in the running example), leftover scheduling has two goals:

- *Minimizing the average JCT* by scheduling tasks from jobs that are closest to completion using Shortest-Remaining-Time-First (SRTF[5]). This schedules the task from stage $S0$ of J_2 in the first time step, leaving 0.21 units of total leftover.

[5] This is the shortest amount of work first, as in Tetris [29].

- *Maximizing efficiency* by packing as many unscheduled tasks as possible – i.e., using Tetris [29]. This results in one more task of stage $S1$ of J_1 to be scheduled, completing leftover redistribution.

3.3.5 Properties of the Offline Scheduler

The former action of the leftover scheduler *enables fast job completions*, while the latter *improves work conservation*. At the same time, because the intra-job scheduler ensures that T_k values are not inflated, the overall scheduling solution maintains the fairness characteristics of DRF and *provides the same performance isolation* and starvation freedom. In summary, the offline CARBYNE scheduler satisfies all the desirable properties.

3.4 From Offline to Online

In the online case, new jobs can arrive before T_k and decrease J_k's resource share, breaking one of the invariants of **Theorem 3.1**. For example, given N jobs and one resource, J_k receives $\frac{1}{N}$-th of the resource; if M more jobs arrive, its share will drop to $\frac{1}{N+M}$-th. If J_k was altruistic earlier, as new jobs arrive, it cannot complete within the previously calculated T_k time units any more.

3.4.1 Analysis of Online Altruistic Scheduling

In practice, even in the online case, we observe marginal impacts on only a handful of jobs (§5.2.3). This is because (i) production clusters run hundreds of jobs in parallel; and (ii) resource requirements of an individual task is significantly smaller than the total capacity of a cluster.

Consider a single resource with total capacity R, and assume that tasks across *all* jobs take the same amount of time (t) and resource ($r \ll R$). Given N jobs, job J_k received $A_k = \frac{1}{N}$-th share of the single resource, and it is expected to complete in T_k time units while scheduling *at most* $\frac{A_k}{r}$ tasks per time unit.

Now consider M new jobs arrive when J_k is $T'_k (< T_K)$ time units away from completion. Their arrival decreases J_k's share to $A'_k = \frac{1}{N+M}$-th, when J_k must be able to schedule tasks at $\frac{A_k}{r}$ rate to complete within T_k.

Assuming all resources being used by the running jobs, the rate at which J_k can schedule tasks is the rate at which tasks finish, i.e., $\frac{t}{R/r}$. Meaning, J_k is expected to take $\frac{A'_k t}{R/r}$ time units to schedule all tasks.

The additional time to schedule the remaining tasks will be negligible compared to remaining time T'_k as long as tasks finish uniformly randomly over time[6] and

$$T'_k \gg \frac{1}{N+M}\frac{r}{R}t$$

The above holds for most production clusters because typically $N \gg 1$ and $R \gg r$.

3.4.2 Bounding Altruism

As a failsafe to prevent jobs from being repetitively punished for altruism, CARBYNE provides a uniformly distributed probability $P(Altruism)$. It dictates with what probability a job will yield its resources during any scheduling event. It is set to 1 by default – i.e., jobs yield resources whenever possible – causing less than 4% of the jobs to suffer at most 20% slowdown (§5.2.3). Decreasing it results in even fewer jobs to suffer slowdown at the expense of lower overall improvements (§5.4.3). In an adversarial environment, one can avoid altruism altogether by setting $P(Altruism) = 0$, which reduces CARBYNE to DRF.

3.5 Discussion

Modern clusters must consider constraints such as data locality and address runtime issues such as stragglers and task failures. CARBYNE is likely to have minimal impact on these aspects.

Data Locality Because disk and memory locality significantly constrain scheduling choices [14, 23, 28], altruistically giving up resources for a data-local task may adversely affect it in the future. However, delay scheduling [54] informs us that waiting even a small amount of time can significantly increase data locality. An altruistically delayed data-local task is likely to find data locality when it is eventually scheduled.

Straggler Mitigation Techniques such as speculative execution are typically employed toward the end of a job to mitigate outliers [13, 15, 23, 56]. CARBYNE is likely to prioritize speculative tasks during leftover scheduling because it selects jobs in the SRTF order.

Handling Task Failures Similar to existing solutions, CARBYNE does not distinguish between new and restarted tasks. However, in case of task failures, CARBYNE must recalculate the completion estimation (T_k) of the corresponding job J_k.

4 Design Details

In this section, we describe how to enable altruism in different cluster schedulers, explain how we have implemented CARBYNE in YARN and Tez, and discuss how we estimate task resource requirements.

4.1 Enabling Altruism in Clusters Schedulers

Enabling altruistic scheduling requires two key components. First, a local *altruistic resource management* module in each application must determine how much resources it can yield. Second, the global cluster scheduler must implement a *leftover resource management* module to reallocate the yielded resources from all applications. RPC mechanisms between the two schedulers may need to be expanded to propagate the new information.

[6] This holds true in large production clusters [54].

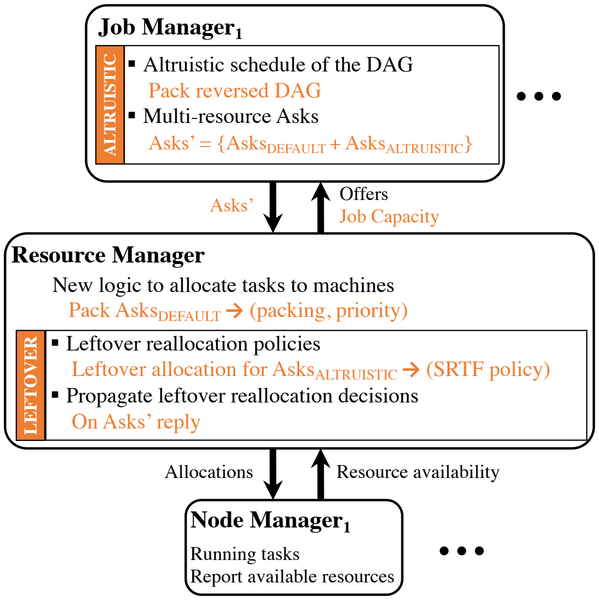

Job Manager$_1$
ALTRUISTIC
Altruistic schedule of the DAG
Pack reversed DAG
Multi-resource Asks
Asks' = {Asks$_{DEFAULT}$ + Asks$_{ALTRUISTIC}$}
Asks'
Offers
Job Capacity
Resource Manager
New logic to allocate tasks to machines
Pack Asks$_{DEFAULT}$ → (packing, priority)
LEFTOVER
Leftover reallocation policies
Leftover allocation for Asks$_{ALTRUISTIC}$ → (SRTF policy)
Propagate leftover reallocation decisions
On Asks' reply
Allocations
Resource availability
Node Manager$_1$
Running tasks
Report available resources

Figure 3: Multi-resource scheduling in a data-parallel cluster. CARBYNE-related changes are shown in orange.

In case of monolithic schedulers such as YARN,[7] individual job managers (e.g., Spark/Tez master) implement the altruism module. The leftover management module is implemented in the cluster-wide resource manager. For two-level schedulers such as Mesos [33], Mesos master's global allocation module should manage the leftover resource management, while the altruistic module can be implemented at the framework-level such as Spark or MPI. Similar architectural principles apply for shared-state schedulers (e.g., Omega [48]) too.

4.2 CARBYNE System

In the following, we describe how we have implemented the two modules to provide altruism. Figure 3 depicts the core pieces of typical parallel cluster schedulers today as well as the core new functionality we add to integrate CARBYNE, marked in orange. We modified and added 1400 lines of code to implement CARBYNE on YARN.

Primer on Data-Parallel Cluster Scheduling Typically, today's data-parallel cluster schedulers divide the scheduling procedure into three parts.

A *node manager* (NM) runs on every machine in the cluster, and it is responsible for running tasks and reporting available resources.

For each job, a *job manager* or Application Master (AM) runs on some machine in the cluster and holds job context information regarding various types of tasks to be executed (pending/in-progress/completed), their dependencies (e.g., DAG) and resource requirements (e.g., memory, CPU).

A cluster-wide *resource manager* (RM) receives `Ask` requests from various AMs for their pending tasks to be scheduled and information about the available resources on different machines from NMs through heartbeats. Based on this information and fairness considerations, it assigns tasks to machines. Typically, an `Ask` contains information such as preferences for data locality, the amount of resources required and priorities at which tasks should be executed. Priority is an useful mechanism to enable AMs to encode their preferences to execute one task over the other (e.g., due to ordering in the DAG or failures).

CARBYNE Implementation We have built CARBYNE as an extension to the YARN [51] and Tez [2] frameworks. To implement Pseudocode 1, we made the following modifications:

1. RPC Mechanism We enhanced the RPC protocol between YARN RM and Tez AM to propagate the total resource allocation of a job as computed at RM, according to the fairness policy enforced in the cluster and the current runnable jobs. Also, we extended the `Ask` data structure to support `Asks` of different types (`Asks`$_{\text{DEFAULT}}$ for tasks that it *must* run in order to not be slow down due to altruism and `Asks`$_{\text{ALTRUISTIC}}$ for tasks that it *may* run if the job scheduler tries to use all the allocated resources) as well as other relevant information (e.g., task's demand estimates across multiple dimensions, task's priority, remaining work etc.).

2. Tez Job Manager Given the most recent resource allocation for a job received from the RM, CARBYNE-enabled Tez AM altruistically schedules the remaining DAG (i.e., unscheduled tasks and their dependencies). It implements, the `IntraJobScheduler` procedure from Pseudocode 1 to encode resource requests (`Asks`$_{\text{DEFAULT}}$) for tasks that should be scheduled via altruism. To do that, CARBYNE does reverse packing using [29] to identify the minimum set of tasks that should run as of now while respecting their dependencies, in order to not slow down the expected job completion time. In addition, it computes `Asks`$_{\text{ALTRUISTIC}}$ using the default intra-job scheduler in Tez that decides on a *breadth-first ordering* for scheduling tasks. The *optional* set of tasks are the ones that can be scheduled according to a greedy scheduler and provides to the RM additional information in order to reallocate leftover resources. While we use [29] to do reverse packing and *breadth-first ordering* to compute `Asks`$_{\text{ALTRUISTIC}}$, any other intra-job scheduling technique (e.g., DAGPS [30, 31], Critical Path Method) can be used as well. AM also includes priorities with each task that serve as "hints" to the RM as discussed below.

3. YARN's Resource Manager The scheduling process in YARN RM is triggered whenever an NM reports

[7] Please refer to [48] on why YARN is a monolithic, not two-level, scheduler.

available resources. We updated YARN RM's matching logic to project tasks' resource requests onto available capacity. First, among the runnable jobs, it computes periodically their DRF allocation and propagates the corresponding resource allocation on the next heartbeat response to every AM. (`InterJobScheduler` procedure from Pseudocode 1). Second, it schedules tasks requests from jobs $\texttt{Asks}_{\text{DEFAULT}}$ using similar heuristics as in Tetris [29] to do packing and reduce job completion time. In addition, we encode the priorities of the tasks into these heuristics in a way similar to DAGPS [30, 31] to account for tasks dependencies in the same DAG's job as instructed by the Tez Job Manager. Packing resource requests from $\texttt{Asks}_{\text{DEFAULT}}$ according to the job AM-hinted task priorities enables the RM scheduler to enforce the altruistic schedule computed by each job AM while improving the cluster utilization. If no more $\texttt{Asks}_{\text{DEFAULT}}$ can be satisfied, the RM scheduler satisfies $\texttt{Asks}_{\text{ALTRUISTIC}}$ resource requests from jobs sorted ascending based on the amount of remaining work (emulating SRTF), until no more requests can be satisfied during this scheduling cycle (`LeftOverScheduler` procedure from Pseudocode 1). This approach prefers efficiency as a secondary objective over JCTs; however, their order can be reversed. For example, the RM scheduler can pack resource requests from $\texttt{Asks}_{\text{ALTRUISTIC}}$ and satisfies $\texttt{Asks}_{\text{DEFAULT}}$ based on the amount of remaining work.

4.3 Demand Estimation

CARBYNE relies on estimates of tasks' resource demands – across CPU, memory, disk, and the network – and their durations to make scheduling decisions. Since modern datacenters have zero or small over-subscription factors [3, 49], CARBYNE considers only access link bandwidths between a machine and its ToR switch.[8]

To estimate tasks' demands and durations, CARBYNE leverages well-established techniques such as using history of prior runs for recurring jobs [11, 24, 29] and assuming tasks from the same stage to have similar resource requirements [14, 27, 44]. We note that requiring manual annotation is also possible, and there are some promising efforts to infer task requirements from program analysis [32]; CARBYNE currently does not use such techniques. While CARBYNE performs the best with accurate estimations, we have found that using the aforementioned techniques work well in practice (§5.4.2).

5 Evaluation

We evaluated CARBYNE on a 100-machine cluster [6] using publicly available benchmarks – TPC-DS, TPC-H, and BigBench – as well as Hive traces collected from large production clusters at Microsoft and Facebook. To understand performance at a larger scale, we used a trace-driven simulator to replay task logs from the same traces. Our key findings are:

- CARBYNE can closely approximate DRF, Tetris, and SJF in terms of fairness, efficiency, and performance, respectively, in both offline and online scenarios (§5.2). Moreover, it provides 1.26× better efficiency and 1.59× lower average completion time than DRF.
- CARBYNE provides similar benefits in large-scale simulations, even for simple MapReduce jobs (§5.3).
- Sensitivity analysis show that CARBYNE performs even better with resource contention, and it is robust to misestimations in resource requirements and when jobs are not always altruistic (§5.4).

In the following, unless otherwise explicitly mentioned, we refer to CARBYNE as our implementation atop YARN and TEZ as described in Section 4 using Tetris as an intra-job scheduler (other than Section 5.5) and DRF for fairness.

5.1 Experimental Setup

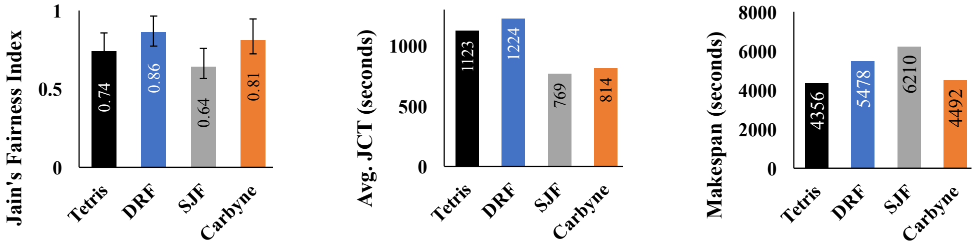
Workloads Our workloads consist of mix of jobs from public benchmarks – TPC-DS [7], TPC-H [8], BigBench [4], and from Microsoft and Facebook production traces collected from clusters with thousands of machines. Our experimental methodology is adapted from and similar to prior work [30, 31]. In each experiment run, the jobs are randomly chosen from one of the corresponding benchmark and follows a Poisson arrival distribution with average inter-arrival time of 20 seconds. Each job lasts from few minutes to tens of minutes, and we generate corresponding input sizes from GBs to hundreds of GBs. Unless otherwise noted, each experiment has 250 jobs. Microsoft workload has 30000 job DAGs with millions of tasks (§2.2). Facebook workload has 7000 jobs and $650,000$ tasks spanning six hours, and jobs use the actual arrival times from the trace. Each experiment is run three times, and we present the median.

Cluster Our experiments use 100 bare-metal servers. Each machine has 20 cores, 128 GB of memory, 128 GB SSD, a 10 Gbps NIC and runs CentOS 7. Meaning, the cluster can run up to 2000 tasks in parallel. The network has a congestion-free core.

Simulator To run CARBYNE at a larger scale and to gain more insights into CARBYNE's performance, we built a simulator that replays job traces. The simulator mimics various aspects of the logs, handling jobs with different arrival times and dependencies as well as multiple fairness and scheduling schemes.

Compared Scheduler Baselines We compare CARBYNE primarily against the following approaches:

[8] For oversubscribed networks with well-defined bottlenecks (e.g., host-toToR links), one may consider bandwidth demands on the oversubscribed links instead.



(a) Inter-job fairness

(b) Job performance

(c) Cluster efficiency

Figure 4: [Cluster] CARBYNE's performance against the best schemes in fairness (DRF), improvement in average JCT (SJF), and achieving cluster efficiency (Tetris) in the *offline* case as shown in Figure 1. CARBYNE approaches the best in each category and outperforms the rest. Higher is better in (a), while the opposite is true in (b) and (c).

1. **DRF**: YARN's implementation of the DRF algorithm [27] for inter-job scheduling along with the default intra-job scheduler in Tez that decides on a *breadth-first ordering* for scheduling tasks;
2. **Tetris**: Uses breadth-first-ordering for intra-job scheduling and Tetris [29] (with its fairness knob, $f \rightarrow 0$) for inter-job scheduling;
3. **SJF**: A shortest-job-first scheduler that uses Critical Path Method (CPM) to determine job duration and schedules job in the shortest-first order. We use SJF primarily as an upper-bound of CARBYNE's improvement in average job completion time.

Metrics Our primary metric to quantify performance is the improvement in the *average JCT*, computed as:

$$\text{Factor of Improvement} = \frac{\text{Duration of an Approach}}{\text{Duration of CARBYNE}} \quad (1)$$

Factor of improvement greater than 1 means CARBYNE is performing better, and vice versa.

Additionally, we use *makespan*, i.e., when all jobs in a workload completed, to measure efficiency.

Finally, to quantify fairness, we compute *Jain's fairness index* [37] on 60 seconds time window intervals, and we plot the average, minimum, and maximum values across all intervals until the workload completes.

5.2 CARBYNE in Testbed Experiments

In this section, we compare CARBYNE to the state-of-the-art solutions across multiple benchmarks and metrics, evaluate its impact on workloads as a whole and on individual jobs, dissect the sources of improvements, and show that CARBYNE has small amortized overheads.

5.2.1 Performance vs. Efficiency vs. Fairness

The Offline Case Figure 4 depicts fairness, the average JCT, and cluster efficiency in our cluster experiments on the TPC-DS workload in the offline case.

We observe that DRF is the most fair approach, with a fairness index of 0.86 on average. However, CARBYNE is only 0.05 units away. In comparison, Tetris is off by 0.12 units and SJF by 0.22. CARBYNE can be unfair on small time intervals due to leftover reallocation, during which jobs can get more or less than their fair share. However, on longer time intervals (60 seconds in our experiments), it is closest to DRF because of its long-term approach toward fairness.

The average JCT is improved the most by SJF, and CARBYNE provides only 1.06× worse average JCT. In comparison, Tetris and DRF are off by 1.46× and 1.59× on average, respectively. Although CARBYNE performs SRTF during leftover reallocation, it is slightly worse than SJF because it attempts not to delay any job beyond its fair-share-calculated completion time.

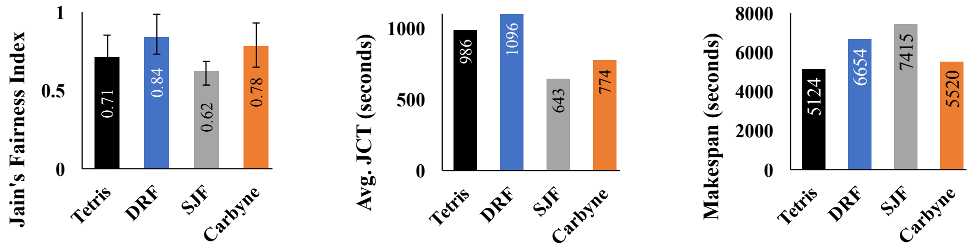
Finally, Tetris provides the highest cluster efficiency (lowest makespan) by efficiently packing tasks across jobs, and CARBYNE is the closest scheme, which is only 1.03× worse than Tetris. Although CARBYNE is also packing tasks, it favors tasks from altruistic jobs instead of treating all runnable tasks equally. In comparison, DRF and SJF are off by 1.26× and 1.43×, respectively.

The Online Case Next, we focus on the case when jobs arrive over time (Section 5.1 has details on arrival process). Figure 5 shows that even in the online case, CARBYNE can closely match DRF (0.06 units worse), SJF (1.2× worse), and Tetris (1.07× worse) in fairness, performance, and efficiency, respectively – although the margins are slightly larger than that in the offline case.

Next, we investigate the root causes behind CARBYNE performing worse in the online case and whether they match our hypotheses in Section 3.4.

5.2.2 JCT Improvements Across Entire Workloads

While CARBYNE performs well for different metrics, the most important metric from a user's perspective is the average JCT. To better understand CARBYNE's impact on JCT, we compare it against popular alternatives: DRF and Tetris (§5.1). Note that we do not include SJF from hereon because it performs the worst both in terms of ef-



(a) Inter-job fairness

(b) Job performance

(c) Cluster efficiency

Figure 5: [Cluster] CARBYNE's performance against the best schemes in achieving fairness (DRF), improvement in average JCT (SJF), and cluster efficiency (Tetris) in the *online* case. CARBYNE approaches the best in each category and outperforms the rest.

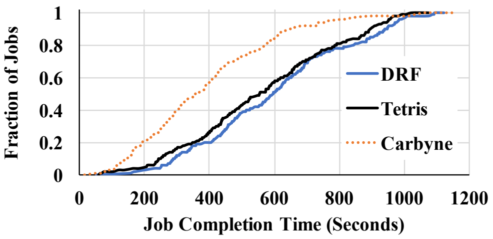


Figure 6: [Cluster] CDF of job completion times using different approaches on TPC-DS workload.

ficiency and fairness, while only marginally outperforming CARBYNE in performance.

Figure 6 shows the distributions of job completion times of the compared approaches for the TPC-DS workload. Only two highest percentiles are worse off by at most 1.1× than Tetris using CARBYNE (not visible in Figure 6). Table 3 shows the corresponding improvement factors for multiple workloads.

CARBYNE vs. DRF For TPC-DS, CARBYNE speeds up jobs by 1.36× on average and 1.88× at the 95th percentile over DRF. Improvement factors are about the same for TPC-H and BigBench workloads. However, corresponding 75th and 95th percentile improvements were up to 1.62× and 1.96× for TPC-H. These gains are mostly due to the presence of a larger fraction of shorter jobs in the TPC-H workload, which benefit more due to CARBYNE's leftover allocation procedure.

CARBYNE vs. Tetris CARBYNE's improvements are similar against Tetris. Tetris ignores jobs dependencies and strives to pack, looking only at the current set of runnable tasks. In contrast, CARBYNE packs critical tasks during the leftover allocation.

5.2.3 Sources of Improvements

We have shown that workloads experience aggregate improvements using CARBYNE. A natural question is then to ask: where do these gains come from? To better answer that question, Figure 7 presents the factors of improvements of individual jobs.

	25th percentile		50th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	1.15	1.12	1.36	1.32
TPC-H	1.11	1.14	1.33	1.29
BigBench	1.13	1.10	1.41	1.35
	75th percentile		95th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	1.55	1.47	1.88	1.75
TPC-H	1.62	1.44	1.96	1.71
BigBench	1.57	1.52	1.85	1.82

Table 3: [Cluster] Factors of improvement across various workloads w.r.t. DRF and Tetris.

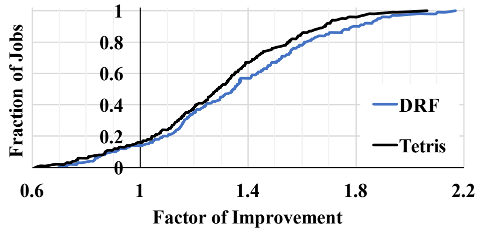


Figure 7: [Cluster] CDF of factors of improvement of individual jobs using CARBYNE w.r.t. different approaches.

We observe that for more than 84% of the jobs, CARBYNE performs significantly better than the alternatives. Only 16% of the jobs slow down – by at most 0.62× – using CARBYNE w.r.t. different approaches. No more than 4% of the jobs slow down more than 0.8×.

Dissecting a Job's Execution To better understand how CARBYNE works in practice, we snapshot the execution of the same job from one of our TPC-DS cluster runs, with and without CARBYNE. Figure 8 presents the number of running tasks during the job execution when running CARBYNE and DRF. In both cases, the jobs were scheduled almost at the same time, approximately 300 seconds after our experiment has started.

The key takeaways are the following. First, in DRF, the breadth-first intra-job scheduler is greedily scheduling tasks whenever it has containers allocated (either due

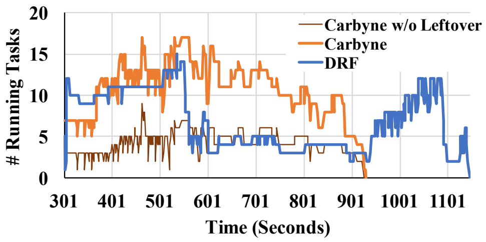


Figure 8: [Cluster] Snapshot of the execution of a TPC-DS query. The job switches from being altruistic in the earlier part of execution to receiving altruisms in the latter part, leading to faster completion. The gap between light and dark orange lines represent the tasks received from leftover allocation.

to fairness or work conservation mechanisms). However, being greedy does not necessarily help. Between times 570 and 940, its progress is slowed down mostly due to high contention in the cluster. Second, CARBYNE's directive of being altruistic helps in the long run. We observe that even when resources are available for allocation (interval between 300 and 550), its allocation is smoother than DRF (dark orange line). Instead of up to 11 tasks to be allocated, it decides to schedule up to 5 while giving up the remaining resources to the leftover. However, note that it does receive back some of the leftover resources (the gap between the light and dark orange lines in Figure 8). Finally, as the job nears completion, CARBYNE provides more leftover resources even when there is high contention in the cluster. In the interval between 570 seconds to 850 seconds, it is receiving significantly larger share than DRF by relying on other jobs' altruisms (i.e., the gap between the dark and light orange lines).

Which Jobs Slow Down? We observed that typically jobs with more work (many tasks and large resource demands per task) are more prone to losses, especially if they contend with many small jobs. One such example is a job that was slowed down by $0.64\times$ w.r.t DRF; it was contending with more than 40 jobs during its lifetime, all of them had less work to do, and hence, got favored by our leftover resource allocation policy.

However, unlike SJF or SRTF, CARBYNE is not inherently biased towards improving shorter jobs at the disadvantage of larger jobs. In fact, half of the jobs that slowed down more than $0.8\times$ (i.e., 2% of all jobs) were small.

We also attempted to bin/categorize the jobs based on characteristics such as the number of tasks, total amount of work, and DAG depth; however, but we did not observe any correlations w.r.t changes in JCT. Instead, we found that whether and how much a job may suffer is a function of the entire cluster's conditions during the job's runtime: available resources, rates of jobs arriving/finishing, their amount of remaining work, and the level of altruism. For better understand their impacts, we perform extensive simulations under multiple parameter choices in Section 5.4.

5.2.4 Scheduling Overheads

Recall from Section 4.2 that CARBYNE expands the Tez AM with an additional altruistic scheduling logic, which is triggered whenever the job's share allocation is changing. We find that CARBYNE-related changes inflate the decision logic by no more than 10 milliseconds, with a negligible increase in memory usage of the AM.

CARBYNE's logic to match tasks to machines – RM's matching logic happens on every heartbeat from NM to RM – is more complex than that in YARN. To quantify its overhead we compute the average time to process heartbeats from NM to RM for different number of pending tasks. We find that for 10000 pending tasks, CARBYNE's additional overhead is 2 milliseconds compared to Tetris (18 ms), and the overhead is up to 4 milliseconds for 50000 pending tasks. CARBYNE also extends the `Ask` requests from AM for multiple resource requirements and encapsulates `Asks` for the altruistic decisions it makes. However, because `Asks` are cumulative, we found that the additional overhead is negligible.

5.3 Performance in Trace-Driven Simulations

To better understand how CARBYNE performs under various parameter choices, we simulate altruistic scheduling using TPC-DS, TPC-H, and BigBench benchmarks as well as Microsoft and Facebook traces.

5.3.1 Benchmarks' Performance in Simulations

To evaluate the fidelity of our simulator, first we replayed the TPC-DS trace logs from cluster experiments in simulation. Table 4 shows the factors of improvement in JCT for TPC-DS workload in simulation that are consistent with that from our cluster experiments (Table 3). Similar results are obtained for the other workloads.

CARBYNE improves over the alternatives by up to $1.59\times$ on average and $7.67\times$ at the 95th percentile. Note that in simulation CARBYNE's improvements are slightly better than that in practice at higher percentiles. This is mainly due to natural factors such as failures, stragglers, and other delays that are not captured by our simulator.

5.3.2 Large-Scale Simulation on Production Trace

Table 4 also shows improvements for a 3000 (10000)-machine Facebook (Microsoft) production trace. CARBYNE outperforms others by up to $2.85\times$ ($8.88\times$) and $2.23\times$ ($7.86\times$) on average (95th percentile).

Note that the gains in production traces are significantly larger than that for other benchmarks. The primary reason is the increased opportunities for temporal rearrangements, both due to more jobs and more machines. In the Facebook trace, all the jobs are MapReduce jobs, and a significant fraction are Map-only jobs. Because

	25th percentile		50th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	1.17	1.12	1.59	1.47
Facebook	1.24	1.24	2.75	2.85
Microsoft	1.16	1.12	2.23	1.74
	75th percentile		95th percentile	
Workload	**DRF**	**Tetris**	**DRF**	**Tetris**
TPC-DS	2.73	2.23	7.67	6.17
Facebook	4.31	4.31	8.77	8.88
Microsoft	3.67	3.28	7.86	7.05

Table 4: [Simulation] Factors of improvement across various workloads w.r.t. DRF and Tetris.

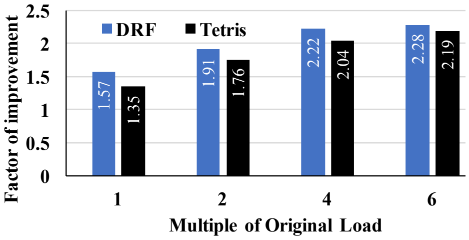


Figure 9: [Simulation] CARBYNE improvements over the alternatives for different cluster loads.

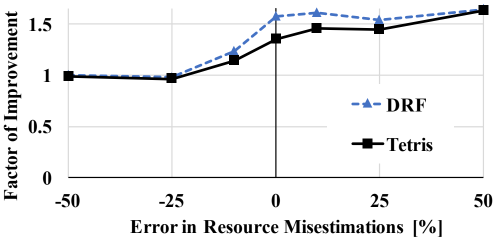


Figure 10: [Simulation] CARBYNE's improvements in terms of average JCT in the presence of misestimations in tasks' resource demands for different intra-job schedulers.

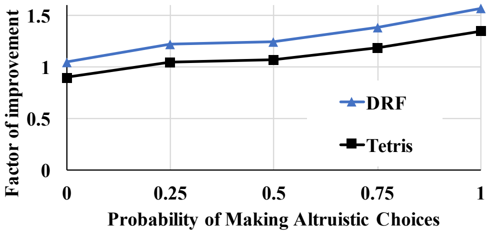


Figure 11: [Simulation] Benefits over the alternatives increase as CARBYNE makes altruistic choices more often. By default, CARBYNE uses 1; i.e., it is altruistic whenever possible.

these jobs only have one or two stages, tasks are less constrained. On the other hand, jobs in Microsoft trace are more complex DAGs with barriers which enables many opportunities for altruism. Furthermore, many jobs are large and have tens to thousands of tasks that can run in parallel. Together, they open up many opportunities for CARBYNE to use the leftover resources.

5.4 Sensitivity Analysis

5.4.1 Impact of Contention

In order to examine CARBYNE performance at different levels of contention, we vary load by changing the number of servers while keeping the workload constant. For example, half as many servers leads to twice as much load on the cluster. Figure 9 shows our results. At $1\times$ cluster load, CARBYNE improves over alternatives by up to $1.57\times$ on average. However, as we increase resource contention, CARBYNE's gains keep increasing. For example, at $2\times$ the load, its gains are between $1.76\times$ and $1.91\times$ on average. This is expected because the more the contention, the better is CARBYNE in carefully rearranging tasks over time. The performance gap increases even more at $4\times$ load. However, at $6\times$, CARBYNE's improvements are almost the same as that at $4\times$ load; this is because the cluster became saturated, without much room for leftover allocations.

5.4.2 Impact of Misestimations

CARBYNE assumes that we can accurately estimate tasks' resource demands and durations. However, accurate estimations can be challenging in practice, and misestimations are inevitable. To understand the impact of misestimations on CARBYNE, we introduce $X\%$ errors in our estimated demands and durations. Specifically, we select $X \in [-50, 50]$, and decrease/increase task resource demands by $\text{task}_{\text{newReq}} = (1 + X/100) * \text{task}_{\text{origReq}}$; task durations are changed accordingly.

Figure 10 shows CARBYNE's performance for varying degrees of estimation errors. It performs consistently better and becomes comparable when we underestimate, because underestimations encourage higher parallelism (due to false sense of abundance), disregarding altruistic scheduling. We also selectively introduced errors only to some of the tasks (not shown) with similar results: CARBYNE outperformed the rest despite misestimations.

5.4.3 Impact of Altruism Levels

The core idea behind CARBYNE is that jobs should altruistically give up resources that do not improve their JCT. In this experiment, we study how different levels of altruism ($P(Altruism)$) impact CARBYNE's benefits.

We observe in Figure 11 that increasing levels of altruism increases CARBYNE's advantage over the alternatives – CARBYNE outperforms them by up to $1.24\times$ when jobs are altruistic half the time ($P(Altruism) = 0.5$) and up to $1.56\times$ at $P(Altruism) = 1$. Last but not the least, at $P(Altruism) = 0$, CARBYNE is comparable to its alternatives; meaning, despite jobs being aggressive in their scheduling, our mechanism is still robust.

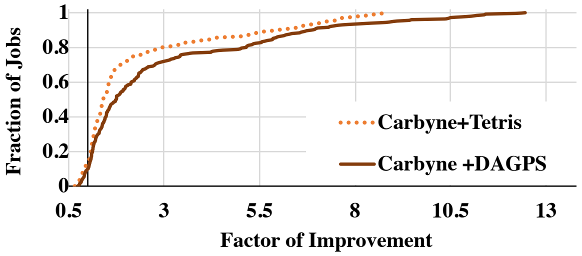


Figure 12: [Simulation] CDF of factors of improvement of individual jobs using CARBYNE + Tetris [29] and CARBYNE + DAGPS [30, 31] w.r.t. DRF. Benefits increase when CARBYNE uses a better DAG scheduler.

5.5 Impact of a Better DAG Scheduler

We also performed simulations where CARBYNE uses DAGPS [30, 31], a more sophisticated intra-job scheduler that considers the entire DAG and its dependencies between stages, instead of Tetris. Figure 12 shows that DAGPS can further increase CARBYNE's performance. We observe that for at least 50% of the jobs, CARBYNE + DAGPS performs better than CARBYNE + Tetris and factors of improvement increase from 2.36× to 3.41× for at least 25% of the jobs w.r.t. DRF. Similar observations hold when comparing with Tetris instead of DRF as the baseline. The additional gains are mainly due to DAGPS's ability to extract more leftover resources without affecting individual DAG completion times.

5.6 Comparison To Multi-Objective Schedulers

As mentioned in Section 1, prior solutions have also attempted to simultaneously meet multiple objectives. For example, Tetris [29] combines heuristics that improve cluster efficiency with those that lower average job completion time, and provides a knob (f) to trade-off fairness for performance. So far, we have shown CARBYNE's benefits against Tetris optimized for performance ($f \rightarrow 0$). Now we compare CARBYNE with Tetris offering strict fairness ($f \rightarrow 1$); we refer to this as $\text{Tetris}_{\text{Fair}}$. We found CARBYNE to be more fair than $\text{Tetris}_{\text{Fair}}$, with the average Jain's fairness index of 0.89 instead of 0.84. This is due to CARBYNE's ability to adhere strictly to fair allocations than $\text{Tetris}_{\text{Fair}}$'s fairness knob. CARBYNE also improves the average job completion time by up to 1.22× and 2.9× at the $95th$ percentile. Although CARBYNE's altruistic behavior to delay tasks in time play a significant role in getting these benefits, $\text{Tetris}_{\text{Fair}}$'s strategy to adopt strict fairness also limits the number of running jobs considered for efficient scheduling, which further hurts job completion times. Finally, these gains have direct implications on cluster efficiency, where CARBYNE outperforms $\text{Tetris}_{\text{Fair}}$ by 1.06×.

6 Related Work

Inter- and Intra-Job Schedulers Traditional cluster resource managers, e.g., Mesos [33] and YARN [51], employ inter-job schedulers [15, 21, 26, 27, 29, 35, 56] to optimize different objectives. For example, DRF [27] for fairness, shortest-job-first (SJF) [26] for minimizing the average JCT, and Tetris [29] for improving efficiency/utilization. More importantly, all of them focus on quick convergence to their preferred metric. Given some share of the resources, intra-job/task schedulers within each job optimize their own completion times. Examples include schedulers that process stages in a breadth-first order [2, 55], ones that follow the critical path [39, 40], and packers [29]. Again, all these schedulers are greedy in that they use up all resources allocated to them.

CARBYNE differs from all of them in at least two ways. First, CARBYNE takes an altruistic approach to maximize resources that can be redistributed. Second, by better redistributing the leftover resources, CARBYNE can simultaneously improve multiple objectives.

Altruistic Schedulers Delay scheduling for data locality [54] and coflow scheduling for network scheduling [20, 21] come the closest to CARBYNE in the high-level principle of altruism. The former waits to get better data locality, while the latter slows individual network flows down so that they all finish together. CARBYNE takes the next step by applying altruistic scheduling in the context of multi-resource scheduling. We leverage altruism to simultaneously improve multiple contrasting objectives.

Leftover Redistribution Hierarchical schedulers in both networking [50] and cluster computing [16] face the same problem as CARBYNE in terms of how to redistribute leftover resources. However, in those cases, entities do not voluntarily yield resources; they only yield resources after saturating their needs. Furthermore, they redistribute by fairly dividing resources among siblings in the hierarchy, whereas CARBYNE takes advantage of leftover resources to improve the average JCT and resource utilization without violating fairness.

DAG and Workflow Schedulers When the entire DAG with the completion times of all stages are known, the Critical Path Method (CPM) [39, 40] is one of the best known algorithms to minimize end-to-end completion times. However, it can be applied only as an intra-job scheduler. Many dynamic heuristics exist for online intra-DAG scheduling with varying results [53]. However, for multiple DAGS, i.e., for inter-DAG scheduling, existing solutions rely on either fair or shortest-first scheduling disciplines. In contrast, CARBYNE combines packing, ordering, and fair allocation.

7 Conclusion

Given the tradeoffs between fairness, performance, and efficiency, modern cluster schedulers [9, 10, 17, 33, 48, 51, 52] focus on performance isolation through *instantaneous* fairness and relegate performance and efficiency as best-effort, secondary goals. However, users perceive isolation only after jobs complete. As long as job completion times do not change, we can take a *long-term*, *altruistic* view instead of an instantaneous one. Using CARBYNE, jobs yield fractions of their resources without inflating their completion times. By combining and rescheduling these leftover resources from collective altruisms, CARBYNE significantly improves application-level performance and cluster utilization while providing the same level of performance isolation as modern schedulers. Deployments and large-scale simulations on benchmarks and production traces show that CARBYNE closely approximates DRF in terms of performance isolation, while providing 1.26× better efficiency and 1.59× lower average completion time.

Acknowledgments

We would like to thank the anonymous OSDI and SIGCOMM reviewers and our shepherd, Phil Levis, for their insightful comments and feedback that helped improve the paper. Robert and Aditya were supported in part by National Science Foundation (grants CNS-1302041, CNS-1330308, and CNS-1345249), Google and the Wisconsin Institute of Software-Defined Datacenters of Madison. Mosharaf was supported in part by National Science Foundation (grants CCF-1629397 and CNS-1563095) and Google.

References

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

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

[3] AWS Innovation at Scale. https://www.youtube.com/watch?v=JIQETrFC_SQ.

[4] Big-Data-Benchmark-for-Big-Bench. https://github.com/intel-hadoop/Big-Data-Benchmark-for-Big-Bench.

[5] Capacity Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[6] Chameleon. https://www.chameleoncloud.org/.

[7] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[8] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch.

[9] YARN Capacity Scheduler. http://goo.gl/cqwcp5.

[10] YARN Fair Scheduler. http://goo.gl/w5edEQ.

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

[12] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[14] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[15] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[16] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *SoCC*, 2013.

[17] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *OSDI*, 2014.

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

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

[20] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[21] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[22] E. G. Coffman and J. L. Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[24] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Eurosys*, pages 99–112, 2012.

[25] M. Garey and D. Johnson. “Strong” NP-completeness results: Motivation, examples, and implications. *Journal of the ACM*, 25(3):499–508, 1978.

[26] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[27] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[28] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *EuroSys*, 2013.

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

[30] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Do the hard stuff first: Scheduling dependent computations in data-analytics clusters. In *MSR-TR-2016-19*, 2016.

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

[32] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[33] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[34] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[35] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[36] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[37] R. Jain, D.-M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Equipment Corporation, 1984.

[38] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, 2012.

[39] J. E. Kelley. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 9(3):296–320, 1961.

[40] J. E. Kelley. The critical-path method: Resources planning and scheduling. *Industrial scheduling*, 13:347–365, 1963.

[41] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[42] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[43] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *UAI*, 2010.

[44] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[45] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataow system. In *SOSP*, 2013.

[46] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond Dominant Resource Fairness: Extensions, limitations, and indivisibilities. In *EC*, 2012.

[47] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

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

[49] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *SIGCOMM*, 2015.

[50] I. Stoica, H. Zhang, and T. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM*, 1997.

[51] V. K. Vavilapalli, A. C. 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 *SoCC*, 2013.

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

[53] J. Yu, R. Buyya, and K. Ramamohanarao. Workflow scheduling algorithms for grid computing. In *Metaheuristics for Scheduling in Distributed Computing Environments*, pages 173–214. 2008.

[54] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[55] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[56] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.