Graphene: Packing and Dependency-aware Scheduling for Data-Parallel Clusters

Robert Grandl[†+], Srikanth Kandula[†], Sriram Rao[†], Aditya Akella[†+], Janardhan Kulkarni[†]
Microsoft[†] and University of Wisconsin-Madison[+]

Abstract– We present a new cluster scheduler, Graphene, aimed at jobs that have a complex dependency structure and heterogeneous resource demands. Relaxing either of these challenges, i.e., scheduling a DAG of homogeneous tasks or an independent set of heterogeneous tasks, leads to NP-hard problems. Reasonable heuristics exist for these simpler problems, but they perform poorly when scheduling heterogeneous DAGs. Our key insights are: (1) focus on the long-running tasks and those with tough-to-pack resource demands, (2) compute a DAG schedule, offline, by first scheduling such troublesome tasks and then scheduling the remaining tasks without violating dependencies. These offline schedules are distilled to a simple precedence order and are enforced by an online component that scales to many jobs. The online component also uses heuristics to compactly pack tasks and to trade-off fairness for faster job completion. Evaluation on a 200-server cluster and using traces of production DAGs at Microsoft, shows that Graphene improves median job completion time by 25% and cluster throughput by 30%.

1 Introduction

Heterogeneous DAGs are increasingly common in data-parallel clusters. We use DAG to refer to a directed acyclic graph where each vertex represents a task and edges encode input-output dependencies. Programming models such as Dryad, SparkSQL and Tez compile user scripts into job DAGs [2, 19, 24, 43, 57, 67]. Our study of a large production cluster in Microsoft shows that jobs have large and complex DAGs; the median DAG has a depth of five and thousands of tasks. Furthermore, there is a substantial variation in task durations (sub-second to hundreds of seconds) and the resource usage of tasks (e.g., compute, memory, network and disk bandwidth). In this paper, we consider the problem of scheduling such heterogeneous DAGs efficiently.

Given job DAGs and a cluster of machines, a cluster scheduler matches tasks to machines *online*. This matching has tight timing requirements due to the scale of modern clusters. Consequently, schedulers use simple heuristics. The heuristics leave gains on the table because they ignore crucial aspects of the problem. For example, critical path-based schedulers [36] only consider the critical path as determined by predicted task runtime and schedule tasks in the order of their critical path length. When DAGs have many parallel chains, running tasks that use different resources together can lead to a better schedule because it allows more tasks to run at the same time. As another example, multi-resource packers [37] aim to run the maximal number of pending tasks that fit within the available resources. When DAGs are deep, locally optimal choices do not always result in the fastest completion time of the whole DAG. Hence, intuitively, considering both variation in resource demands and dependencies may result in better schedules for heterogeneous DAGs.

By comparing the completion times of jobs in the production cluster with those achieved by an oracle, we estimate that the median job can be sped up by up to 50%. We observe that individual DAGs have fewer tasks running relative to the optimal schedule at some point in their lifetime. The cluster has lower overall utilization because (a) resources are idle even when tasks are pending due to dependencies or resource fragmentation, and (b) fewer jobs are released because users wait for the output of previous jobs. Given the large investment in such clusters, even a modest increase in utilization and job latency can have business impact [1, 10, 61].

We note that the optimal schedule for heterogeneous DAGs is intractable [54, 55]. Prior algorithmic work exists especially on simpler versions of the problem [18, 20, 21, 35, 50, 60, 65]. However, we are yet to find one that holds in the practical setting of a data-parallel cluster. Specifically, the solution has to work online, scale to large and complex DAGs as well as many concurrent jobs, cope with machine-level fragmentation as opposed to imagining one cluster-wide resource pool, and handle multiple objectives such as fairness, latency and throughput.

In this paper, we describe a cluster scheduler Graphene that efficiently schedules heterogeneous DAGs. To identify a good schedule for one DAG, we observe that the pathologically bad schedules in today's approaches mostly arise due to two reasons: (a) long-running tasks have no other work to overlap with them, which reduces parallelism, and (b) the tasks that are runnable do not

pack well with each other, which increases resource fragmentation. Our approach is to identify the potentially *troublesome* tasks, such as those that run for a very long time or are hard to pack, and place them first onto a *virtual resource-time space*. This space would have $d+1$ dimensions when tasks require d resources; the last dimension being time. Our intuition 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.

At job submission time, Graphene builds a preferred schedule for a job as shown in Figure 1. After identifying a subset of troublesome tasks, the remaining tasks are divided into the parent, child and sibling subsets. Graphene first places the troublesome tasks onto a virtual resource-time space and then places the remaining subsets. Realizing this idea has a few challenges. Which choice of troublesome tasks leads to the best schedule? Further, since troublesome tasks are placed first, when a task is considered for placement some of its parent tasks and some of its children tasks may already have been placed in the virtual space. How to guarantee that every task can be placed without violating dependencies? Our answers are in §4.

Graphene's online component schedules the tasks of each DAG in the order of their starting time in the virtual resource-time space. Furthermore, across the many DAGs that may be running in the cluster, the online component respects different objectives– low job latency, high cluster throughput and fairness. These objectives can translate to discordant actions. For example, a fairness scheme such as DRF [33] may want to give resources to a certain job but the shortest-job-first heuristic that reduces job latency may pick a different job. Similarly, the task that is most likely to reduce resource fragmentation [37] may not start early in the virtual resource-time space. Our reconciliation heuristic intuitively picks tasks by consensus (e.g., based on a weighted combination of the scores received by a task from each objective). However, to maintain predictable performance, we limit unfairness to an operator-configured threshold.

We have implemented Graphene as extensions to Apache YARN and Tez and have experimented with jobs from TPC-DS, TPC-H and other benchmarks on a 200 server cluster. Furthermore, we evaluate Graphene in simulations on 20,000 DAGs from a production cluster.

To summarize, our key contributions are:

1. A characterization of the DAGs seen in production at Microsoft and an analysis of the performance of various DAG scheduling algorithms (§2).
2. A novel DAG scheduler that combines multi-resource packing and dependency awareness (§4).
3. An online inter-job scheduler that mimics the preferred per-job schedules while bounding unfairness (§5) for many fairness models [6, 33, 47].

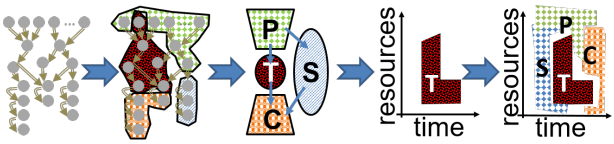


Figure 1: Steps taken by Graphene from a DAG on the left to its schedule on the right. Troublesome tasks T (in red) are placed first. The remaining tasks (parents P, children C and siblings S) are placed *on top* of T in a careful order to ensure compactness and respect dependencies.

4. Using our new lower bound on the completion time of a DAG (§6), we show that the schedules built by Graphene's offline component are within 1.04 times the theoretically optimal schedule (OPT) for half of the production DAGs; three quarters are within 1.13 times and the worst is 1.75 times OPT.
5. Our experiments show that Graphene improves the completion time of half of the DAGs by 19% to 31% across the various workloads. Production DAGs improve relatively more because those DAGs are more complex and have diverse resource demands. The gains accrue from running more tasks at a time; the cluster's job throughput (e.g., makespan) also improves by about 25%.

While we present our work in the context of cluster scheduling, DAGs are a powerful and general abstraction for scheduling problems. Scheduling the network transfers of a multi-way join or the work in a geo-distributed analytics job etc. can be represented as DAGs. We offer early results in §9 from applying Graphene to scheduling the DAGs that arise in distributed build systems [3, 34] and in request-response workflows [46, 66].

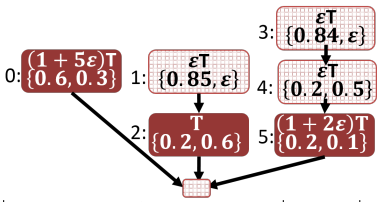
2 Primer on Scheduling Job DAGs

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. Given a set of concurrent jobs $\{\mathcal{G}\}$, the cluster scheduler maps tasks to machines while respecting capacity constraints and task dependencies. Tasks may be allocated fewer resources than their (peak) demands causing them to take longer (task duration is assumed to be a convex function of resource allocation). The goals of a typical cluster scheduler are *high performance* (measured using job throughput, average job completion time and overall cluster utilization) while offering *fairness* (measured w.r.t how resources are divided).

2.2 An illustrative example

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



Technique	Execution Order	Time	Worst-case
OPT	$t_1 \to t_3 \to \{t_4, t_0\} \to \{t_0, t_2, t_5\}$	T	–
CPSched	$t_0 \to t_3 \to t_4 \to t_5 \to t_1 \to t_2$	$3T$	$O(n) \times$ OPT
Tetris	$t_0 \to t_1 \to t_3 \to t_2 \to t_4 \to t_5$	$3T$	$O(d) \times$ OPT

Figure 2: An example DAG where a packer (Tetris [37]) and a Critical Path scheduler take 3× the optimal algo OPT. Here, GRAPHENE is close to OPT (see §2.2). Assume $\varepsilon \to 0$.

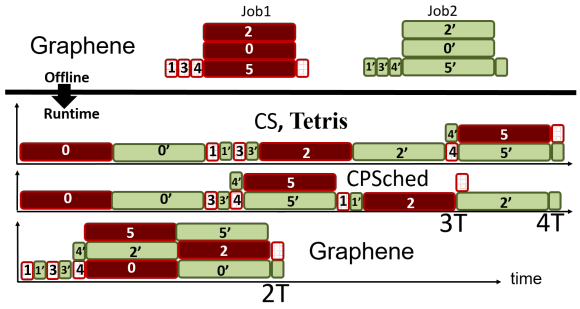


Figure 3: Illustrating the online case. Online, GRAPHENE schedules the tasks of each DAG in the order of their start time in the offline schedule. Resources are to be divided fairly between two jobs both of which have the same DAG shown in Figure 2. Notice that fairness interleaves allocation between the jobs. GRAPHENE's average job completion time and makespan improve by 25% to 75% and 50% to 100% respectively depending on the compared baseline.

node labels represent the duration (top) and the demands for two resources (bottom). Assume that the capacity is 1 for both resources. Let ε represent a value approaching zero.

Intuitively, a good schedule would overlap the long-running tasks shown with a dark background. The resulting optimal schedule (OPT) is listed in the table below the figure. OPT overlaps the execution of all the long-running tasks, t_0, t_2 and t_5, and finishes in T.

Since such long-running or resource-intensive tasks can be present anywhere in the DAG, greedy schedulers often perform poorly as we show next.

Critical path-based schedulers (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 table shows the task execution order with CPSched.[1] CPSched ignores the resources needed by tasks. In this example, CPSched performs poorly because it does not schedule tasks off the critical path early (e.g., t_1, t_3, t_4) even though doing so reduces resource fragmentation by overlapping long-running tasks.

Packers, such as Tetris [37], match tasks to machines so as to maximize the number of simultaneously running tasks. Tetris greedily picks the task with the highest value of the dot product between task's demand vector and the available resource vector. The table 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 though that can lead to a better global packing.

GRAPHENE comes close to the optimal schedule for this example. When searching for troublesome subsets, it will consider the subset $\{t_0, t_2, t_5\}$ 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 ($\{t_1, t_3, t_4\}$) and any children are then placed *before* and *after* the troublesome tasks respectively in a compact manner while maintaining inter-task dependencies.

Online: Consider two jobs that have the DAG shown in Figure 2. Figure 3 illustrates the online schedule when resources are to be divided evenly between these jobs (e.g., slot fairness [13]).

The offline schedule computed by GRAPHENE for each of the jobs, which overlaps the long-running tasks (t_0, t_2, t_5), is shown on top. The online component distills these schedules into a precedence order over tasks. For example, the order for both jobs is: $t_1 \to t_3 \to t_4 \to \{t_0, t_2, t_5\}$. Figure 3, bottom, shows a time-lapse of the task execution.

Capacity Scheduler (CS) [7], a widely used cluster scheduler, checks for which DAG the next available slot has to be allocated, and then picks (in a breadth-first order) a runnable task from the designated DAG that fits the available resources. Figure 3 shows that CS results in an average job completion time (JCT) and makespan of $4T$. Fairness causes the scheduler to interleave the tasks of the two jobs. Tetris happens to produce a similar schedule to CS. Note that this online schedule is far from the preferred per-DAG schedule, only a few of the long-running tasks overlap. Similar to CS, most production schedulers, including Spark, schedule tasks based on some topological ordering of the DAG while using fairness to decide which job to give resources to next. Hence, they behave similarly.

Figure 3 also shows that CPSched has an average JCT and makespan of $3.5T$ and $4T$ respectively. This is because CPSched finishes the t_1 tasks of both the jobs late; because t_1 has a small CP length. Therefore the t_2 tasks from both

[1] CP of t_0, t_1, t_3 is $T(1+5\varepsilon)$, $T(1+\varepsilon)$ and $T(1+4\varepsilon)$ respectively. Tasks can run simultaneously only if their total demand is below capacity.

[2] Tetris' packing score for each task, in descending order, is t_0=0.9, t_1=0.85, t_3=0.84, t_2=0.8, t_4=0.7 and t_5=0.3.

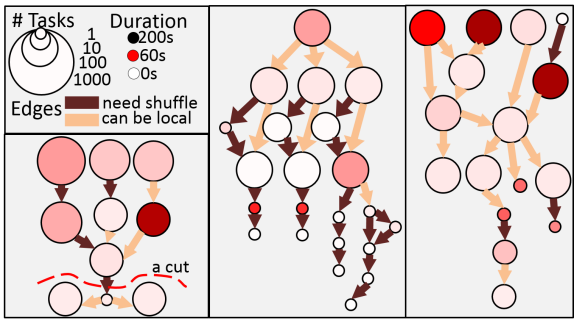


Figure 4: Visualizing a few small production DAGs. The legend is in the top left. See §2.3.

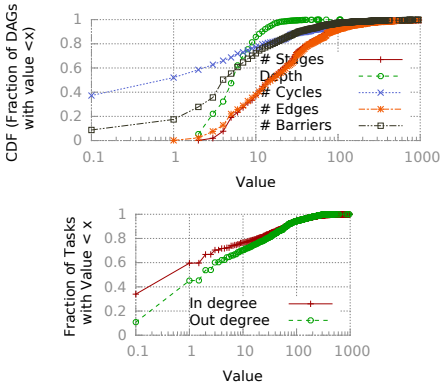


Figure 5: Characterizing structural properties of the DAGs

jobs do not overlap with any other long task.

Finally, the figure shows that Graphene has an average JCT and makespan of $2T$. Graphene achieves this by just enforcing a precedence order within each DAG. In particular, note that all of the schedulers are work-conserving; they leave resources idle only if no schedulable task can fit in those resources. The key difference, among the schedulers, is the *order* in which they consider the tasks for scheduling. Another difference is whether this *order* is computed based only on the runnable tasks (e.g., ordering runnable tasks on their CP length, packing score or on their breadth-first position) versus ordering based on a global optimization. Informally, Graphene's gains arise from looking at the entire DAG and choosing a globally optimal schedule.

2.3 Analyzing DAGs in Production

To understand the problem with actual DAGs and at scale, we examine (a) the production jobs from a cluster of tens of thousands of servers at Microsoft, (b) jobs from a 200 server Hive [67] cluster and (c) jobs from a Condor cluster [5].

Structural properties: As a preliminary, Figure 4 illustrates some production DAGs at Microsoft. Each circle denotes a stage. By stage, we mean a collection of tasks that perform the same computation on different data (e.g. all map tasks). The size of the circle corresponds to the number of tasks in logarithmic scale, the circle's color corresponds to the average task duration in linear scale and the edge color denotes the type of dependency. We see `W`-shaped DAGs (bottom left) that join multiple datasets, inverted `V`-shaped DAGs (middle) that perform different analysis on a dataset, and more complex shapes (right) wherein multiple datasets are analyzed leading to multiple outputs. Note also the varying average durations of the tasks (circle colors); the resource variations are not shown for simplicity. Further, note cycles in the DAGs which are possibly due to self-joins and range-partitions.

Figure 5 plots a CDF of various structural properties of the DAGs from the Microsoft cluster. Since the x-axis is in log scale, we put the probability mass for $x = 0$ at $x = 0.1$.

We see that the median DAG has a depth of five. To compare, a map-reduce job has depth of one. A quarter of the DAGs have depth above ten.

While 40% of the DAGs are trees (i.e., no cycle after ignoring the direction of dependency), we see that many have cycles (half of the DAGs have at least 3 cycles); the average number of tasks in a cycle is 5 (not shown in the figure). Tree-like DAGs are an important special case as they are, theoretically, more tractable to schedule [48].

The DAGs can be quite large; the median job has thousands of tasks and tens of *stages*. To compare, a map-reduce job has two stages. The number of stages that a query translates to depends upon the optimizer. For benchmark queries in TPC-DS and TPC-H, when compared to the DAGs generated by Hive, we see that the query optimizer in this cluster creates DAGs with substantially fewer stages. The dataflow within a task is largely through memory (a task can have multiple relational operators within it). However, the dataflow across stages is harder to optimize since it is unknown when or on which machine the consuming tasks will be scheduled; hence fewer stages, in general, lead to faster execution.

We also see that most of the edges are not barriers (e.g., those labeled "can be local"). Note the gap between the orange stars line and the black squares line in Figure 5 which correspond to counts of all edges and barriers respectively. A barrier edge indicates that every task in the parent stage should finish before any task in the child stage begins.

We observe that DAGs can be cut into portions such that all tasks after the cut can only begin after every task before the cut has finished. An example cut is shown with a red dashed line on the DAG in Figure 4 (left bottom). Cuts often arise because a dataset, perhaps newly generated by upstream tasks, has to be partitioned before downstream tasks can begin. Cuts are convenient because the optimal schedule for the DAG is a concatenation of the optimal schedules of the cut portions of that DAG. We observe that 24% of the production DAGs can be split into four or more parts.

The median (75th percentile) task in-degree and out-

	CPU	Mem.	Network		Disk	
			R	W	R	W
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.

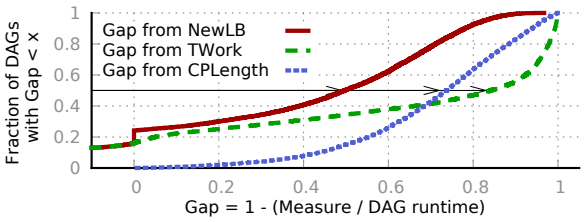


Figure 6: CDF of *gap* between DAG runtime and several measures. Gap is computed as $1 - \frac{\text{measure}}{\text{DAG runtime}}$.

degree are 1 (8) and 3 (20) respectively. For a map-reduce job with m mappers and r reducers, the median in-degree will be 0 if $m \geq r$ and m otherwise. The larger out-degree is because stages that read from the file-system are data reductive; hence, the query optimizer creates fewer downstream tasks overall.

Overall, we conclude that DAGs are both large and have complex structures.

Diversity in resource demands: Similar to prior work [33, 37], we measure the coefficient-of-variation (CoV) across tasks in their demand for various resources. Table 1 shows substantial variability. The variability is possibly due to the 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).

Potential for improvement: To quantify potential gains, we compare the runtime of production DAGs to two measures. The first, `CPLength`, is the duration of the DAG's critical path. If the available parallelism is infinite, the DAG would finish within `CPLength`. The second, `TWork`, is the total work in the DAG normalized by the cluster share of that DAG (a formula is in Table 3.) If there were no dependencies and perfect packing, a DAG would finish within `TWork`. Figure 6 plots a CDF of the relative gap between the runtime of a DAG and these measures. Half of the jobs have a gap of over 70% for both `CPLength` and `TWork`.

Understanding the gap: A careful reader would notice that about 15% of the DAGs finish faster than some measures. This is because our production scheduler occasionally gives jobs more than their fair share if the cluster has spare resources; hence, measures which assume that the cluster share will be the minimum guaranteed for the

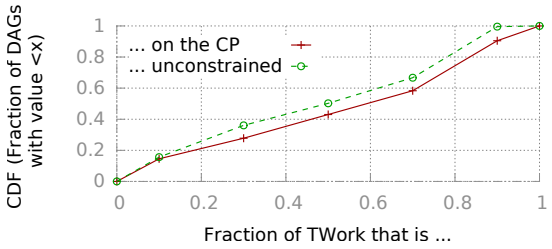


Figure 7: A CDF of the portion of a DAGs `TWork` that is present on the critical path or is "unconstrained" i.e. in tasks with no parents.

DAG can be larger than the actual completion time. We will ignore such DAGs for this analysis.

Suppose `OPT` is the optimal completion time for a DAG given a certain cluster share. We know that actual runtime is larger than `OPT` and that the above measures are smaller than `OPT`. Now, the gap could be due to one of two reasons. (1) The measure is loose (i.e., well below `OPT`). In practice, we found this to be the case because `CPLength` ignores all the work off the critical path and `TWork` ignores dependencies. (2) The observed runtimes of DAGs are inflated by runtime artifacts such as task failures, stragglers and performance interference from other cluster activity [17, 74].

To correct for (2), we discount the effects of runtime artifacts on the above computed DAG runtime as follows. First, we chose the fastest completion time from a group of recurring jobs. It is unlikely that every execution suffers from failures. Second, to correct for stragglers–one or a few tasks holding up job progress–we deduct from completion time the periods when the job ran fewer than ten concurrent tasks. Note that both these changes reduce the gap; hence they under-estimate the potential gain.

Further, to correct for (1), we develop 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 all the tasks in a job stage (e.g., a map or reduce or join) have similar dependencies, durations and resource needs. 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.

Where does the work lie in a DAG? We now focus on the parts of the DAG that do more work (measured as the product of task duration and resource needs). Figure 7 shows a CDF over DAGs of the fraction of work that is on the critical path and in tasks with no parents ("unconstrained"). If all of the work lies on the critical path or in un-constrained tasks, critical path scheduling and packers would perform well. We see from the figure that only about 18% of the DAGs have 80% of their work in unconstrained tasks; the corresponding number for critical path is 23%. We offer this as a motivation for the need of a holistic solution that considers the entire DAG.

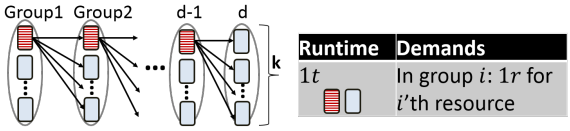


Figure 8: A counter-example DAG that shows any scheduler not considering DAG structure will be $\Omega(d)$ times OPT.

To summarize, (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

Lemma 1 (Dependencies)**.** Any scheduling algorithm, deterministic or randomized, that does not account for the DAG structure (e.g., only schedules currently runnable tasks) is $\Omega(d)$ times OPT where d is the number of resources.

Proof. We prove this first for deterministic schedulers by constructing an adversarial DAG in response to scheduler's actions. Consider the DAG in Figure 8 which is a chain of d *groups* with dependencies going from left to right. All tasks run for $1t$. Each group (oval) has k tasks. Tasks in the i'th group only use the i'th resource and require $1r$. Assume that the capacity for all d resources is $1r$. Hence, if dependencies are handled correctly, it is possible to run up to d tasks together, one from each of the groups. Further, in each group there is a certain task colored red that is the parent of all tasks in the next level. This information is unavailable (and unused) by schedulers that do not consider the DAG structure. Hence, regardless of which order the scheduler picks tasks, the adversary chooses the last task in that group to be the red task. This leads to a schedule that takes kdt time. Observe that OPT only requires $(k+d-1)t$ since it can schedule the red tasks first, one after the other (in $(d-1)t$); now all the blue tasks become runnable and can be scheduled in kt more steps (one task from each group per step). Thus, all deterministic schedulers are $\Omega(d) \times$ OPT.

To extend to randomized algorithms, we use Yao's minimax principle [56]. Specifically, to establish that the lower bound on the expected performance of a randomized algorithm is X, we have to choose some distribution over inputs such that no deterministic algorithm performs better than X in expectation on that input distribution. Suppose each task in a group has the same probability of being red, the best deterministic algorithm on this input has an expected schedule time of $k+k(d-1)t/2$ which is still $\Omega(d) \times$ OPT. □

Lemma 1 applies to the following multi-resource packers [37, 58, 69, 70] since they ignore dependencies.

Lemma 2 (Resource Variation)**.** Schedulers that ignore resource heterogeneity have poor worst-case performance. For example, critical path scheduling can be $\Omega(n)$ times OPT where n is the number of tasks in a DAG.

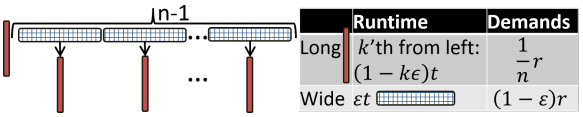


Figure 9: A DAG where critical path scheduling is $O(n)$ times OPT where n is the number of nodes in the DAG.

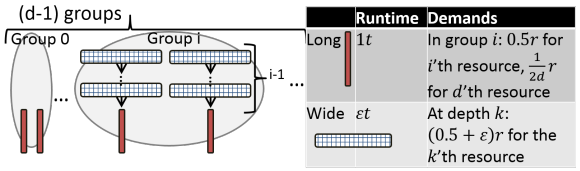


Figure 10: A DAG where Tetris [37] is $2d-2$ times OPT when tasks use d kinds of resources.

Proof. Figure 9 shows an example DAG where CPSched takes n times worse than OPT for a DAG with $2n$ tasks. As before, assume capacity is $1r$. The long tasks have duration $\sim 1t$ and demand $\frac{1}{n}r$ whereas the wide tasks have duration εt and require $(1-\varepsilon)r$. 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. Thus, CPSched is $O(n)$ times OPT. □

Figure 10 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 ignores resource usages.

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). The key intuition here is that DAGs are hard to schedule because of their complex structure and because of discretization issues when tasks use multiple resources (fragmentation, task placement etc.) Graphene is close to OPT on all of the described examples and is within 1.04 times OPT for half of the production DAGs (see §8).

Scheme(s)	DAG annotation needed		
	DAG Structure	Task Resource Demands	Task Durations
Hive [67], Spark [73], Tez [2]	✓‡	✓‡	
Yarn [8], Mesos [41]		✓‡	
CPSched	✓‡		✓‡
Tetris [37]		✓‡	✓‡
Graphene	✓‡	✓‡	✓‡

*: online estimates suffice; ‡: online refinements possible

Table 2: DAG annotations that are required by the various schedulers. Note that Yarn and Mesos are meta-schedulers that are agnostic to the actual work (e.g., DAGs) being scheduled.

2.5 Acquiring annotated DAGs

Acquiring an annotated DAG is non-trivial. Much prior work has similar requirements as Graphene (see Table 2). There are two parts to this: the structure of the DAG and the task profiles (resource needs and durations).

DAG structure: In order to launch a task only after parent tasks finish, every DAG scheduler is aware of the DAG structure. Furthermore, the DAG is often known before the job starts. Runtime changes to the DAG, if they happen, only affect small portions of a DAG. For example, our scheduler adds an aggregation tree in front of a reduce stage depending upon runtime conditions.

Task resource demands and durations: Graphene requires each task to be annotated with the demands for any resource that could be congested; the other resources do not affect scheduling. Here, we consider four resources (cores, memory, disk and network bandwidth). Schedulers such as Yarn, Mesos, Hive and Spark ask users to annotate their tasks with cores and memory requirements; for example, [1 core, 1 GB] is the default in Hadoop 2.6. Graphene requires annotations for more resources as well as the durations of tasks.

There are some early efforts to obtain these profiles (tasks' demands and durations) automatically. For example, in the production cluster at Microsoft, up to 40% of the resources in the examined cluster are used by *recurring* jobs; the same script executes periodically on newly arriving data. Recurring jobs can be identified based on the job name (e.g., `LogMiner_date[_time]`) and prior work shows that the task profiles of these jobs can be estimated from history (after normalizing for the size of input) [16]. For the remaining jobs, some prior work builds profiles via sampling [59], program analysis [40], or based on online observations of the actual usages of tasks in the same *stage* [17]. Our method is described in §7; our sensitivity analysis in §8.4 shows that Graphene is robust to modest amounts of estimation error.

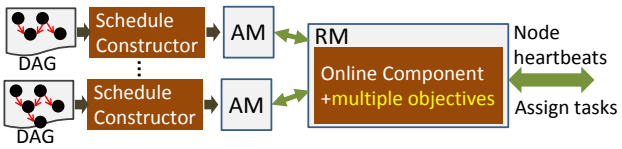


Figure 11: Graphene builds schedules per DAG at job submission. The runtime component handles online aspects. AM and RM refer to the YARN's application and resource manager components.

3 Novel ideas in Graphene

Cluster scheduling is the problem of matching tasks to machines. Most production schedulers today do so in an online manner and have very tight timing constraints since clusters have thousands of servers, many jobs that each have many pending tasks and tasks that finish in seconds or less [8, 73]. Given such stringent time budget, carefully considering large DAGs seems daunting.

As noted in §1, a key design decision in Graphene 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 11 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. While we have no guarantees for our particular division, we note that it scales to large clusters and outperforms the state-of-art in experiments.

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. Furthermore, 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 handled directly during 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

Term	Definition
Task t	an atomic unit of execution
Stage s	a group of tasks that run same code on different data
$\texttt{TWork}(s)$	$\max_r \text{resource}_r \frac{1}{C_r} \sum_{t \in s} t_{\text{duration}} * t^r_{\text{demands}}$
$\texttt{ExecTime}(s)$	estimated time to execute tasks in s
$\mathcal{V}, \mathcal{G}$	$\mathcal{V}$ denotes all stages (and tasks) in a DAG $\mathcal{G}$
$\mathcal{S}$	a virtual schedule: i.e. a placement of a given DAG of tasks in a resource-time space
$C(s,\mathcal{G})$, $P(s,\mathcal{G})$, $D(s,\mathcal{G})$, $A(s,\mathcal{G})$, $U(s,\mathcal{G})$	Children, parents, descendants, ancestors and unordered neighbors of s in $\mathcal{G}$; note that $U(s,\mathcal{G}) = \mathcal{V} - A(s,\mathcal{G}) - D(s,\mathcal{G}) - \{s\}$

Table 3: Glossary of terms.

```
1  Func: BuildSchedule:
2  Input: G: a DAG, m: number of machines
3  Output: An ordered list of tasks t ∈ G
4  Ans ← {}
5  foreach dag G' ∈ CutDAGs(G) do
6      S_best ← ∅ // best schedule for G' thus far
7      foreach sets {T,S,P,C} ∈ CandidateTroublesomeTasks(G') do
8          Space S ← CreateSpace(m) //resource-time space
9          S ← PlaceTasks(T, S, G') // trouble goes first
10         S ← TrySubsetOrders({SCP, SPC, CSP, PSC}, S, G')
11         if S < S_best then S_best ← S //keep the best schedule;
12     Ans ← Ans ∪ OrderTasks(G', S_best) // concatenate schedules
```

Figure 12: Pseudocode for constructing the schedule for a DAG. Helper methods are in Figure 13.

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. One key novel aspect is bounding the extent of unfairness.

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

4 Scheduling one DAG

Graphene builds the schedule for a DAG in three steps. Figure 1 illustrates these steps and Figure 12 has a simplified pseudocode. First, Graphene 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, Graphene carefully restricts the order in which different subsets are placed such that the troublesome tasks go first and there are no dead-ends (§4.3). Graphene picks the most compact schedule after iterating over many choices for troublesome tasks. The resulting schedule is passed on to the online component (§5).

4.1 Searching for troublesome tasks

To identify *troublesome* tasks, Graphene computes two scores per task. The first, `LongScore`, divides the task duration by the maximum 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 Table 3) by the time a greedy packer takes to schedule that stage. Tasks that are more difficult to pack would have a lower `FragScore`. Given thresholds l and f, Graphene picks tasks with $\texttt{LongScore} \geq l$ or $\texttt{FragScore} \leq f$. Intuitively, this biases towards selecting tasks that are more likely to hurt the schedule because they are long or difficult to pack. Each value of $\{l, f\}$ leads to a choice of troublesome tasks `T` which leads to a schedule (after placing the tasks in `T` first and then the other subsets); Graphene iterates over different values for the l and f thresholds and picks the most compact schedule.

To speed up this search, (1) rather than choose the threshold values arbitrarily, Graphene picks values that are discriminative, i.e. those that lead to different choices of troublesome tasks, and (2) Graphene remembers the set of troublesome tasks that were already explored (by previous settings of the thresholds) so that only one schedule is built for each troublesome set. Note also that the different choices of troublesome tasks can be explored in parallel. Further improvements are in §4.4.

As shown in Figure 13 (line 22), 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 `S` consists of the remaining tasks.

4.2 Compactly placing tasks of a subset

Given a subset of tasks and a partially occupied space, how best to pack the tasks while respecting dependencies? Graphene uses the following logic for each of the subsets `T`, `P`, `S` and `C`. 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 different schedules since greedy packing yields different results based on the order in which tasks are placed. Figure 14:PlaceTasks shows some simplified pseudo-code. Traversing the tasks in either placement has $n \log n$ complexity for a subset of n tasks and if there are m machines,

```
Func: CutDAGs:
Input: 𝒢: input DAG Output: ℒ: ordered list of DAGs
ℒ ← {𝒢}
toProcess.push(𝒢)
while ! toProcess.empty() do
    𝒢′ ← toProcess.pop()
    foreach stage s ∈ 𝒢′ do
        if U(s, 𝒢′) = ∅ // no unordered neighbors then
            {𝒢1, 𝒢2} ← {A(s, 𝒢′) ∪ s, D(s, 𝒢′)} // cut at s
            Replace 𝒢′ with {𝒢1, 𝒢2} in ℒ
            toProcess.push(𝒢1)
            toProcess.push(𝒢2)
            break

Func: CandidateTroublesomeTasks:
Input: DAG 𝒢; Output: list ℒ of sets T,S,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)/ExecTime(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); S ← 𝒱 − T − P − C;
        ℒ ← ℒ ∪ {T,S,P,C}
```

Figure 13: Identifying various candidates for troublesome tasks and dividing the DAG into four subsets.

placing tasks greedily has $n\log(mn)$ complexity.

4.3 Subset orders that guarantee feasibility

For each division of DAG into subsets T, S, P, C, Graphene considers these 4 orders: TSCP, TSPC, TPSC or TCSP. That is, in the TSCP order, it first places all tasks in T, then tasks in S, then tasks in C and finally all tasks in P. Intuitively, this helps because the troublesome tasks T are always placed first. Further, other orders may lead to dead-ends. For example, consider the order TPCS; by the time some task t in the subset S is considered for placement, parents of t and children of t may already have been placed since they may belong to the sets P and C respectively. Hence, it may be impossible to place t without violating dependencies. We prove that the above orders avoid *dead-ends* and are the only orders beginning with T to do so.

Note also that only one of the forwards or backwards placements (described 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 noted above, the forwards placement places a task after its last finishing ancestor but ignores descendants and can hence violate dependencies if used for P; because by definition every task in the parent subset P has at least one descendant task. Analogously, tasks in C cannot be placed *backwards*. Tasks in S 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; details are in Figure 14:TrySubsetOrders. We prove the following:

```
Func: PlaceTasks(𝒱, 𝒮, 𝒢):
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
return min(PlaceTasksF(𝒱, 𝒮, 𝒢), PlaceTasksB(𝒱, 𝒮, 𝒢))

Func: PlaceTasksF: // forwards placement, inputs and outputs same as PlaceTasks
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) ⊆ F}
    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: PlaceTasksB: // backwards placement, inputs and outputs same as PlaceTasks
Input: 𝒱, 𝒮, 𝒢; Output: 𝒮
𝒮 ← Clone(𝒮)
finished placement set F ← {v ∈ 𝒢 | v already placed in 𝒮}
while true do
    ready set R ← {v ∈ 𝒱 − F | C(v, 𝒢) ⊆ F}
    if R = ∅ then break // all done;
    v′ ← task in R with longest runtime
    t ← (min_{v∈C(v,𝒢)} BeginTime(v, 𝒮)) − v′.duration
    // place v′ at the latest time ≤ t when its resource demands can be met
    F ← F ∪ v′

Func: TrySubsetOrders:
Input: 𝒢: a DAG, 𝒮_in: space with tasks in T already placed
Output: A space that has the 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(S, 𝒮1, 𝒢), 𝒢), 𝒢), //SPC
PlaceTasksB(P, PlaceTasksF(C, PlaceTasks(S, 𝒮2, 𝒢), 𝒢), 𝒢), //SCP
PlaceTasksB(P, PlaceTasksB(S, PlaceTasksF(C, 𝒮3, 𝒢), 𝒢), 𝒢), //CSP
PlaceTasksF(C, PlaceTasksF(S, PlaceTasksB(P, 𝒮4, 𝒢), 𝒢), 𝒢) //PSC
);
```

Figure 14: Pseudocode for the description in §4.2, §4.3.

Lemma 3. (Correctness) Our method in §4.1–§4.3 satisfies dependencies and avoids *dead-ends*. (Completeness) The method explores every order that places troublesome tasks first and is free of *dead-ends*.

Intuitively, the proof (omitted for space) follows from (1) all four subsets are closed and hence intra-subset dependencies are respected by the placement logic in §4.2 whether in the forward or in the backward placement, (2) the inter-subset orders and the corresponding restrictions to only use forwards and/or backwards placements specified in §4.3 ensure that dependencies across subsets are respected and, (3) every other order that begins with T can lead to dead-ends.

4.4 Enhancements

We note a few enhancements. First, as noted in §2.3, 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 the complexity of schedule construction. We recursively cut until no more cuts are possible. Figure 13:CutDAGs shows how to do this in linear time. 24% of the production DAGs can be split into four or more parts. Second, and along similar lines, whenever possible we reduce com-

```
Func: FindAppropriateTasksForMachine:
Input: m: vector of available resources at machine; 𝒥: set of jobs
  with task details {t_duration, t_demands, t_priScore}; deficit:
  counters for fairness;
Parameters: κ: unfairness bound; rp: remote penalty
Output: S, the set of tasks to be allocated on the machine
S ← ∅
while true do
    foreach task t do
        {pScore_t, oScore_t} ← {0, 0}
        rPenalty_t ← t is locality sensitive ? rp : 1
        if t_demands ≤ m // fits? then
            pScore_t ← (m · t_demands) rPenalty_t // dot product
        else
            // what-if analysis: "overbook or wait".
            ∀tasks t' affected by t running at m, let before(t'),
            after(t') be expected completion times before and
            after placing t at m
            benefit = nextSchedOpp + t_duration − after(t)
            cost = Σ_aff. tasks t' (after(t') − before(t'))
            if benefit > cost then oScore_t = benefit − cost;
        job j ∋ t, srpt_j ← Σ_pending u ∈ j u_duration * |u_demands|
        perfScore_t ← t_priScore {pScore_t, oScore_t} − η srpt_j
    t^best ← arg max{perfScore_t | t} // task with highest perf score
    if t^best = ∅ then break // no new task can be scheduled on this
      machine;
    g' ← jobgroup with highest deficit counter
    if deficit_g' ≥ κC then t^best ← arg max{perfScore_t | t ∈ g'};
    S ← S ∪ t^best
    m ← [m − t^best_demands]_0+
    deficit_g ← deficit_g +
      factor(t^best_demands) * { fairShare_g − 1   t ∈ jobgroup g
                               { fairShare_g       otherwise
```

Figure 15: Simplified pseudocode for the online component.

plexity by reasoning over stages. Stages are collections of tasks and are 10 to 10^3 times fewer in number than tasks. Third, schedule computation can be sped up in a few ways. Parallelizing the search will help the most, i.e. examine different choices for troublesome tasks T in parallel. Working over more compact representations (e.g., scaling down the DAG and the cluster by a corresponding amount) will also help. Fourth, jobs that are short-lived, or only use a small amount of resources, or do not have complex DAG structures, will bypass the offline portion of Graphene. Fifth, the complexity of schedule construction is independent of the sizes of the subsets T, S, P, C that Graphene divides the DAG into. However, if |T| is very large, the approach of placing troublesome tasks first and other tasks carefully around them is unlikely to help. We prune such choices of T without further exploration. Among the schedules built by Graphene for production DAGs, the median DAG has 17% of its tasks considered troublesome; these tasks contribute to 32% of the work in that job. Finally, note that it is possible to recursively employ this logic: i.e., given a DAG $\mathcal{G}$, pick a troublesome subset T, let $\mathcal{G}'$ be the sub-DAG over tasks in T, repeat the logic on $\mathcal{G}'$. We defer further examination of this approach to future work.

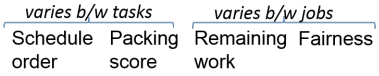


Figure 16: The various aspects considered by Graphene's online component when matching tasks to machines.

5 Scheduling many DAGs

Given the preferred schedules for each job, we describe how the Graphene inter-job scheduler matches tasks to machines online. Recall the example in Figure 3. The scheduling procedure is triggered when a machine m reports its vector of available resources to the cluster-wide resource manager. Given a set of runnable jobs (and their tasks), the scheduler returns a list of tasks to be allocated on that machine. The challenge is to enforce the per-job order computed in §4 while also packing tasks for cluster efficiency, ensuring low JCTs, and enforcing fairness.

5.1 Inter-job Scheduler

Enforcing preferred schedules. Using the per-DAG schedule constructed in §4, a $t_{priScore}$ is computed for each task t by (1) ranking tasks in increasing order of their start time in the schedule and (2) dividing the rank by the number of tasks in the DAG so that the result is between 1 (for the task that begins first) and 0. As noted below, Graphene preferentially schedules tasks with a higher $t_{priScore}$ value first.

Packing efficiency. Graphene borrows ideas from [37] to improve packing efficiency. For every task, it computes a packing score $\texttt{pScore}_t$ as a dot product between the task demand vector and the machine's available resource vector. To favor local placement, when remote resources are needed, $\texttt{pScore}_t$ is reduced by multiplying with a remote penalty rp ($\in [0,1]$). Sensitivity analysis on the value of rp is in §8.4.

Job completion time. Graphene estimates the remaining work in a job j similar to [37]; $\texttt{srpt}_j$ is a sum over the remaining tasks to schedule in j, the product of their duration and resource demands. A lower score implies less work remaining in the job j.

Bounding unfairness. Graphene trades off fairness for better performance while ensuring that the maximum unfairness is below an operator configured threshold. Specifically, Graphene maintains deficit counters [64] across jobs to measure unfairness. The deficit counters are updated as follows. When a task t from a group g is scheduled, its deficit increases by $\texttt{factor}_t \times (\texttt{fairShare}_g - 1)$ and the deficit of all the other groups g' increases by $\texttt{factor}_t \times \texttt{fairShare}_{g'}$. This update lowers the deficit counter of g proportional to the resources allocated to it and increases the deficit counters of other groups to remember that they were treated unfairly. Further, by varying the value of $\texttt{factor}_t$, Graphene can support different fairness schemes: e.g., $\texttt{factor}_t = 1$ mimics slot fairness and $\texttt{factor}_t$ = *demand of the dominant resource of g*

mimics DRF [33].

Combining schedule order, packing, completion time and fairness. Graphene attempts to simultaneously consider the above four aspects; as shown in Figure 16, some of the aspects vary with the task while others vary across jobs. First, Graphene combines the performance related aspects into a single score, i.e., $\texttt{perfScore}_t = \texttt{pScore}_t \cdot \texttt{t}_{priScore} - \eta\texttt{srpt}_j$. η is a parameter that is automatically updated based on the average srpt and pScore across jobs and tasks. Subtracting $\eta \cdot \texttt{srpt}_j$ prefers shorter jobs. Sensitivity analysis on the value of η is in §8.4. Intuitively, the combined value $\texttt{perfScore}_t$ softly enforces the various objectives. For example, if a task t 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 $\texttt{perfScore}_t$. When the objectives are discordant, colloquially, the task preferred by a majority of objectives t will have the highest $\texttt{perfScore}_t$.

Next, to trade-off performance while bounding unfairness, let the most unfairly treated group (the one with the highest deficit counter) be g_{unfair}. If the deficit counter of g_{unfair} is below the unfairness threshold, then Graphene picks the task with the maximum perfScore from among all groups; else it picks the task with the maximum perfScore from g_{unfair}. The unfairness threshold is κC where κ (< 1) is a tunable parameter and C is the cluster capacity.

Further details, including a pseudo-code, are in Figure 15.

Judicious overbooking: We observe that over-allocating some resources by a small amount, which we call *overbooking* can improve throughput. For example, suppose that tasks with a duration t require $0.6r$. Running two tasks instead of one, i.e. over-booking by 20%, improves the task throughput from 1 task/t to 2 tasks/$1.2t$. Note however, that this holds only if the resource type is such that over-allocation leads to a graceful degradation. In the above example, we assumed that when demand exceeds capacity, the total goodput remains intact and the net effect is that tasks take proportionately longer. This holds only for certain resources (such as network bandwidth as opposed to memory which will lead to a net slow-down due to thrashing) and for small amounts of overbooking (so as to not trigger collapse issues such as incast).

A key tussle with overbooking is that while it improves throughput it may hurt latency, because the runtimes of all tasks executing on the overbooked machine will increase. Worse, if resources will become free at some other machine soon, then overbooking may be counterproductive. For the above example, if another machine can run the second task at time $+\varepsilon$. Without overbooking, tasks will finish at $\{t, t+\varepsilon\}$ and with overbooking both finish at $1.2t$. Hence, optimal overbooking is NP-hard [53].

Graphene offers a heuristic for overbooking. First, it uses micro-benchmarks to determine how task runtimes will be delayed when each resource is overbooked. These functions are concave and vary across resource types. Next, per potential task to overbook, Graphene runs a what-if analysis to decide between overbooking and waiting. We compute the expected completion times of all affected tasks after overbooking. Note that resource overbooking delays these tasks but the extent of delay can vary. The benefit of overbooking is how much earlier would the new task finish with overbooking versus having to wait for next-free-resource. The cost is the increase in runtime of all the other tasks due to overbooking. Thus, overbooking score equals benefit – cost.

Putting the above description into one place, Figure 15 offers simplified pseudo-code for the online component of Graphene.

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 they either ignore all the work off the critical path (e.g., CPLen) or ignore dependencies and assume perfect packing (e.g., TWork). Since the optimal solution is intractable to compute [54, 55], without a good lower bound, it is hard to assess the quality of a heuristic solution such as Graphene.

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. There are four key ideas. First, 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. Notice that the maximum of several lower bounds is also a valid lower bound. 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 $\texttt{ModCP}_G$ formula in Equation 1c where one stage s along any path p is replaced with the total work in that stage. A third idea is that some stages have all-to-all dependencies to all parents and children. That is all its tasks have to finish *after* the last parent task finishes and *before* the first child task can start. For such stages, we can replace them with their total work. To see why this helps, consider a stage of n tasks with duration d and demand vector $\mathbf{r}$. This stage will now contribute $\max(nd\frac{\mathbf{r}}{C}, d)$ instead of d. When n or $\mathbf{r}$ are large, this leads to a larger CPLength. 34% of the stages in our production DAGs have this property. Finally, we group tasks having identical parents and children even though their

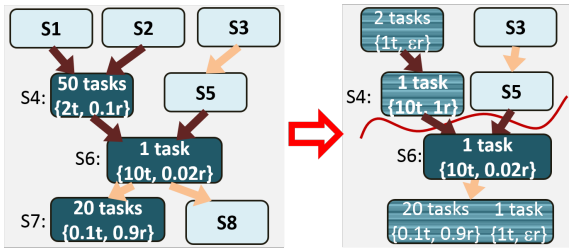
$$\texttt{CPLen}_{\mathcal{G}} = \max_{\text{path } p \in \mathcal{G}} \sum_{\text{task } t \in p} t_{\text{duration}} \tag{1a}$$

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

$$\texttt{ModCP}_{\mathcal{G}} = \max_{\text{path } p \in \mathcal{G}} \max_{\text{stage } s \in p} \left(\max(\texttt{TWork}_s, \texttt{CPLen}_s) + \sum_{g \in p - \{s\}} \min_{\text{task } t \in g} t_{\text{duration}} \right) \tag{1c}$$

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

Figure 17: 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.



Unmarked stages have 1 task @ {1t, εr}

Lower-bounds: **CPLen** = 14t, **Twork** = 12t and **NewLB**= 22.8t
OPT = 24t

Figure 18: For the DAG on left, the modified DAG used by our lower bound is on the right. Lower bound improves from $14t$ to $22.8t$. Edge types are as per the legend in Figure 4.

functions differ. For example, two map stages preceding the same reduce stage (in a join). Larger groups let us cumulatively account for their tasks which helps the second and third changes above.

The take-away is that the new lower bound NewLB is much tighter and allows us to show that Graphene is close to OPT; since by definition of a lower bound Graphene ≥ OPT ≥ NewLB.

We have the following lemma:

Lemma 4. NewLB (Eqn. 1d) is a valid lower bound for DAG runtime, and NewLB ≥ max(CPLen, TWork).

Proof. First, observe that the maximum of the lower bounds is also a lower bound. Second, observe that by definition the DAG is cut into parts that have no overlap. Hence, the lower bound of the DAG is equal to the sum of the lower bounds of the parts. This supports Eqn. 1d.

Next, grouping stages with identical parent and child stages is appropriate because (a) there are no dependencies between these tasks and (b) the definition of a stage has been a group of independent tasks that can run in parallel and adhere to a specific dependence pattern with tasks in parent and child stages.

Using the total work to be done in a stage instead of the duration of a single task is appropriate. This holds because either all of the work has to be done in line (when all parents and children have all-to-all edges) or at least one stage on a path through the DAG will have to finish all of its work. This supports Eqn. 1c.

Stepping back, this new lower bound was possible because of the abundance of groups of independent tasks that is common in data-parallel DAGs. As we did not relax either dependence satisfaction or resource capacity limits, this lower bound is much tighter than other bounds based on linear programs that relax those aspects [22, 52, 53]. □

Illustrative Example: Figure 18 shows an example DAG and the various lower bounds. Note that, CPLen is $14t$; the longest critical path is $\{S_1, S_2\} \rightarrow S_4 \rightarrow S_6 \rightarrow S_8$. Further, TWork is $12t$ with stages S_4, S_6, S_7 contributing $10t, 0.2t, 1.8t$ respectively; as before, we assume the cluster capacity C is $1r$.

The DAG on the right shows the modifications used by Graphene. First, we can cut at S_6 per logic in §4.4; all tasks in previous stages should finish, because of the barriers between S_4, S_5 and S_6. Second, we can replace S_4 with its duration of $10t$ because (a) S_4 has to fully finish since its parents S_1, S_2 and its children S_6 are connected with a barrier and (b) the fastest schedule for S_4 would run ten tasks in each wave and five waves leading to a duration of $10t$. Third, for the portion of the DAG below the cut, we can replace at most one stage with its modified duration; we choose to do that for the stage created by merging S_7 and S_8 (they have the same parent and children) and their duration together is $1.8t$. Finally, we can also merge S_1 and S_2 but their duration remains $1t$ because of their small resource requirement and this does not affect the overall bound. Hence, the overall lower bound is now $22.8t$; $11t$ from CPLen of the sub-DAG above the cut, $10t$ from the duration of S_6 and $1.8t$ from the modified duration of $\{S_7, S_8\}$. Due to our ideas, the bound has improved from $14t$ to $22.8t$.

In this case, OPT is $24t$. The gap between NewLB and OPT arises from two causes here. (a) A $1t$ addition in the sub-DAG above the cut because S_5 cannot overlap with $\{S_1, S_2\}$ (due to dependencies; S_3 has to finish) and also cannot overlap with S_4 (because S_4 fully uses capacity of $1r$). (b) A $0.2t$ increase in the sub-DAG below the cut; because due to fragmentation only one task of S_7 can run at a time causing it to take twenty waves and finish in $2t$ as opposed to the $1.8t$ predicted by perfect packing. Note however that NewLB is much closer to OPT.

7 Graphene 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 (unoptimized) schedule constructor finishes in tens of seconds on the DAGs used in experiments; this is in the same ballpark as the time to compile and

query-optimize these DAGs. Recurring jobs use previously constructed schedules. Each DAG is managed by an instance of the Tez AM which closely resembles other frameworks such as FlumeJava [25] and Dryad [43]. 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 Graphene, 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. Here, we describe some key aspects.

7.1 DAG Annotations

Recall from §2.5 that Graphene requires a more detailed annotation of DAGs than existing systems: specifically, it needs task durations and estimates of network and disk usages; the usages of cores and memory are already available [8, 67, 73].

Our approach is to construct estimates for the average task in each stage using a combination of historical data and prediction. These estimates are used by the offline portion of Graphene (§4). As noticed by prior work, recurring jobs are common in our production clusters and historical usages, after normalizing for the change in data volume, are predictive for such job groups [16]. The online portion of Graphene (§5) refines these estimates based on the actual work of a task (e.g., by noting its input size) and based on the executions of earlier tasks; since (a) tasks in the same stage often run in multiple waves due to capacity limits and (b) running tasks issue periodic progress reports [8, 17].

In our evaluation, we execute the jobs once and use the actual observed usage (from job history) to compute the necessary annotations. We normalize both the duration and usage estimates by the tasks' input size, as appropriate. A sensitivity analysis that introduces different amounts of error to the estimates and shows their effect on performance is in §8.4.

We observe that Graphene is rather robust to estimation error because relatively small differences in tasks' duration and usages do not change the schedule. For example, while it is useful to know that reduce and join tasks are network-heavy as opposed to map tasks which have no network usage, it is less useful to know precisely how much network usage a reducer or a join task will have; the actual usage would vary, at runtime, in any case due to contention, thread or process scheduling, etc. Similarly, while it is useful to know that tasks in a certain stage will take ten times longer, on average, and hence it is better to overlap those tasks with unrelated work, it is less useful to know the exact duration of a task; again, the exact durations will vary because of contention, machine-specific slowdowns etc. [17].

7.2 Efficient online matching

Naively implementing our runtime component (§5) would improve schedule quality at the cost of delaying scheduling. We use *bundling* to offset this issue.

Some background: The matching logic in typical schedulers is heartbeat-based [8]. When a machine heartbeats to the RM, the allocator (0) maintains an ordering over pending tasks, (1) picks the first 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.

A naive implementation of the runtime component would examine all the pending tasks; thereby increasing the time to match.

Instead, we propose to bundle the allocations. Specifically, rather than breaking the loop after finding the first schedulable task (step 1 above), we keep along a *bundle* of tasks that can all be potentially scheduled on the machine. At the end of one pass, we assign multiple tasks by choosing from among those in the bundle.

The *bundle* amortizes the cost of examining the pending tasks. We can allocate multiple tasks in one pass as opposed to one pass per task. It is also easy to see that bundling admits non-greedy choices and that the pass can be terminated early when the bundle has good-enough tasks. We have refactored the Yarn scheduler with configurable choices for (1) which tasks to add to the bundle, (2) when to terminate bundling and (3) which tasks to pick from the bundle. From conversations with Hadoop committers, these code-changes help improve matching efficiency and code readability.

7.3 Co-existing with other features

We note that a cluster scheduler performs other roles besides matching tasks to machines. Several of these roles such as handling outliers and failed tasks differently [17, 74], delay scheduling [72], reservations [12, 30] or supporting heterogeneous clusters where only some servers may have GPUs [11] are implemented as preconditions to the main schedule loop, i.e. they are checked first, or are implemented by partitioning the tasks that will be considered in the scheduling loop. Since Graphene's changes only affect the inner core of the schedule loop (e.g., given a set of pending tasks, which subset to allocate to a machine), our implementation co-exists with these features.

8 Evaluation

Our key evaluation results are as follows.

(1) In experiments on a 200 server cluster, relative to `Tez` jobs running on `YARN`, Graphene improves completion time of half of the jobs by 19% to 31% across various benchmarks. 25% of the jobs improve by 30% to 49%. The extent of gains depends on the workload (complexity of DAGs,

resource usage variations etc.).

(2) On over 20,000 DAGs from production clusters, the schedules constructed by Graphene 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.

(3) By examining further details, we show that the gains are from better packing dependent tasks. Makespan (and cluster throughput) improve by a similar amount. More resources are used, on average, by Graphene and trading off short-term unfairness improves performance.

(4) We also compare with several alternative schedulers and offer a sensitivity analysis to cluster load, various parameter choices, and annotation errors.

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 [39].

Workload: Our workload mix consists of jobs from public benchmarks—TPC-H [14], TPC-DS [15], BigBench [4], 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, job arrival is modeled as 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 query finishes in under ten minutes on the idle cluster. A typical experiment run has about 120 jobs. The results presented are the median over three runs.

Compared Schemes: We experimentally compare Graphene with 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]. To tease apart the gains from the offline and online components, we also offer results for (4) `Tez+G+CS` and (5) `Tez+G+Tetris` which use the offline constructed schedules at the job manager (to request containers in that order) but the online components are agnostic to the desired schedule (either the default capacity scheduler or Tetris respectively). Using simulations, we also compare Graphene against the following schemes: (6) `BFS` : breadth first order, (7) `CP` : critical path order, (8) `Random` order, (9) `StripPart` [20], (10) `Tetris` [37], and (11) `CoffmanGraham` [29].

All of the above schemes except (9) are work-conserving. (6)–(8) and (10) pick only from among the runnable tasks but vary in the specific heuristic. (9) and (11) perform more complex schedule construction, as we will discuss later.

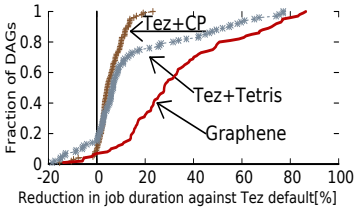
Metrics: Improvement in JCT is our key metric. Between two schemes, we measure the *normalized gap* in JCTs. That is, the difference in the runtime of a job divided by the job runtime; the normalization lets us compare jobs with very different runtimes. We also measure makespan, i.e., the time to finish a given set of jobs, Jain's fairness index [45], and the actual usages of various resources in the cluster.

8.2 How does Graphene do in experiments?

Job Completion Time: Relative to `Tez`, Figure 19 shows that Graphene improves half of the DAGs by 19 to 31%; the extent of gains depends on the workload and varies across benchmarks. A quarter of the DAGs improve by 30 to 49%. We see occasional regressions. Up to 5% of the jobs in the TPC-DS benchmark slow down with Graphene; the maximum slowdown is 16%. We found this to primarily happen on the shorter jobs and believe it is due to noise from runtime artifacts such as stragglers and task failures [17]. The table in Figure 19 shows the results for all the 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 larger gains.

Relative to the alternatives, Figure 19 shows that Graphene 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 Graphene because Tetris' packing logic reduces fragmentation. 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 substantially improves DAG completion time.

Where do the gains come from? Figure 20 offers more detail on an example experimental run. Graphene keeps more tasks running on the cluster and hence finishes faster (Figure 20a). The other schemes take over 20% longer. Graphene runs more tasks by reducing fragmentation and by overbooking resources such as network and disk that do not lose goodput when demand exceeds capacity (unlike say memory). Comparing Figure 20b with Figures 20c, 20d, the average allocation of all resources is higher with Graphene. Occasionally, Graphene allocates over 100% of the network and disk. One caveat about our measurement methodology here: we take the peak usage of a task and assume that the task held on to those resources for the entirety of its lifetime; hence, the us-



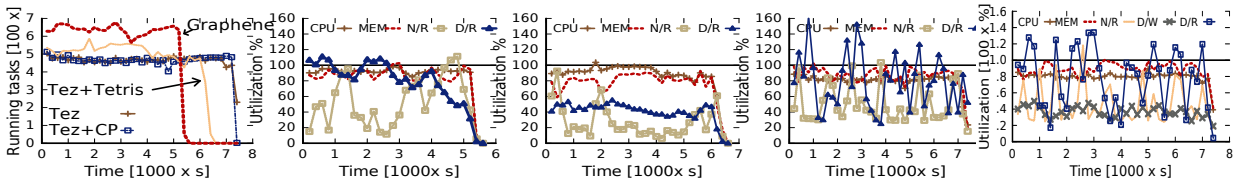
(a) CDF jobs on TPC-DS workload

Workload	25th % T+CP	25th % T+T	25th % G	50th % T+CP	50th % T+T	50th % G	75th % T+CP	75th % T+T	75th % G
TPC-DS	2.0	1.9	**16.0**	4.1	6.5	**27.8**	8.9	16.6	**45.7**
TPC-H	1.8	1.5	7.6	3.8	8.9	30.5	7.7	15.0	48.3
BigBench	4.1	2.0	5.6	6.4	6.2	25.0	21.7	18.5	33.3
MS-Prod	-3.0	3.2	4.4	1.0	5.8	19.0	4.5	14.2	29.7

G is GRAPHENE, **T+T** is Tez + Tetris and **T+CP** is Tez + CP. The improvements are relative to Tez. Each group of columns reads out the gaps at the percentile in the label of that group.

(b) Improvements in JCT across all the workloads

Figure 19: Comparing completion time improvements of various schemes relative to Tez.



(a) Running tasks (b) GRAPHENE (c) Tez + Tetris (d) Tez + CP (e) Tez

Figure 20: 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.

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

Table 4: 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
	GRAPHENE	+2%	0.72	0.83	0.89

Table 5: Fairness: Shows the performance gap and Jain's fairness index when used with 2 queues (even share) versus 1 queue. Here, a score of 1 indicates perfect fairness.

ages are over-estimates for all schemes. Tez + Tetris, the closest alternative, has fewer tasks running at all times because (a) it does not overbook (resource usages are below 100% in Figure 20c) and (b) it has a worse global packing for a DAG because it ignores dependencies and packs only the runnable tasks. 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. Overall, GRAPHENE gains by increasing the task throughput.

Makespan: To evaluate makespan, we make one change to the experiment setup– all jobs arrive within the first few minutes. Everything else remains the same. Table 4 shows the gap in makespan for different cases. Due to careful packing, GRAPHENE sustains high cluster resource utilization which in turn enables jobs to finish quickly: makespan improves 30% relative to Tez and over 20% relative to alternatives.

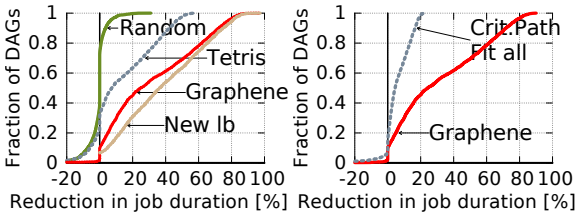
Fairness: Can we improve performance while also being fair? Intuitively, fairness may hurt performance since fairly dividing resources may lower overall utilization or slow-down some jobs. To evaluate fairness, we make one change to the experiment set up. The jobs are evenly and randomly distributed among two queues and the scheduler has to divide resources evenly.

	Tez + G + CS 50th %ile	Tez + G + CS 75th %ile	Tez + G + Tetris 50th %ile	Tez + G + Tetris 75th %ile
JCT	28.1	41.7	26.3	39.5
Makespan	24.6	35.4	22.5	32.2

Table 6: Shows the performance gap, relative to GRAPHENE, when the preferred schedules of DAGs are used by the job managers but ignored by the cluster scheduler.

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

Value of enforcing preferred schedules online: Recall that GRAPHENE's online component enforces the preferred schedules constructed by the offline component. To tease apart the value of this combination, we consider alternatives wherein the job managers use the preferred schedules (to request containers in that order) but the cluster scheduler is agnostic; i.e. it simply runs the default



(a) GRAPHENE vs. Baselines (b) GRAPHENE vs. Alternates

Figure 21: Comparing GRAPHENE with other schemes. We removed the lines for CG and StripPart from the right figure because they hug $x = 0$; see Table 7.

		25^{th}	50^{th}	75^{th}	90^{th}
GRAPHENE		7	25	57	74
Random		−2	0	1	4
Crit.Path	Fit cpu/mem	−2	0	2	1
	Fit all	1	4	13	16
	Overbooking	2	9	24	31
Tetris	Fit all	0	7	29	42
	Overbooking	0	11	33	49
Strip Part.	Fit all	0	1	12	27
	Overbooking	0	2	16	33
Coffman-Graham.	Fit all	0	1	12	26
	Fit cpu/mem	−2	0	0	2

Table 7: Reading out the gaps from Figure 21. The improvements are relative to BreadthFirst - standard approach used in Tez.

capacity scheduler or Tetris (we call these Tez + G + CS and Tez + G + Tetris respectively). Table 6 shows that GRAPHENE offers 26% and 28% better median JCT compared to Tez + G + Tetris and Tez + G + CS. This experiment was conducted on a smaller 50 server cluster with different hardware so these numbers are not directly comparable with the remaining experiments; we offer them merely as a qualitative validation of GRAPHENE's combination of online and offline components.

8.3 Comparing with alternatives

We use simulations to compare a wider set of 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.

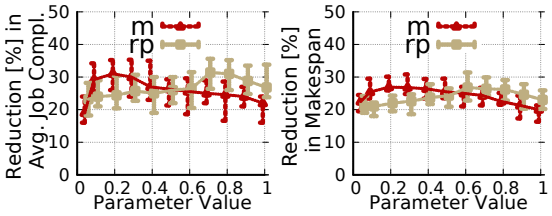
Figure 21 compares the schedules constructed by GRAPHENE with the schedules from other algorithms. Table 7 reads out the gaps at various percentiles. We observe that GRAPHENE's gains at the end of schedule construction are about the same as those at runtime (Figure 19). This is interesting because the 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 is made up by the gains from better packing (across DAGs) and trading off some short-term unfairness.

Second, GRAPHENE's gains are considerable compared to the alternatives. CP and Tetris are the closest. The reason is that GRAPHENE 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 [29]) is at most 2 times optimal. However, it does not perform well on the heterogeneous DAGs seen in production. Some recent extensions of CG to handle heterogeneity ignore fragmentation when resources are divided across machines [49].

Fourth, StripPart [20] combines resource packing and task dependencies and has the best-known approximation ratio: $O(\log n)$ on a DAG with n tasks [20]. The key idea is to partition tasks into *levels* such that all dependencies go across levels and then to tightly pack each level. We find that StripPart under-performs simpler heuristics in practice because (a) independent tasks that happen to fall into different levels cannot be packed together leading to wasted resources between levels and (b) the recommended per-level packers (e.g. [60]) do not support multiple resources and vector packing [58].

How close is GRAPHENE to Optimal? Comparing GRAPHENE with NewLB, we find that GRAPHENE is optimal for about 40% of the DAGs. For half (three quarters) of the DAGs, GRAPHENE is within 4% (13%) of the new lower bound. A gap still remains: for 10% of DAGs, GRAPHENE takes 25% longer. Manually examining these DAGs shows that NewLB is loose for most of them (deriving a tighter lower bound is an open problem). In sum, GRAPHENE is close to optimal for most of the production DAGs.



(a) Job Duration (b) Makespan

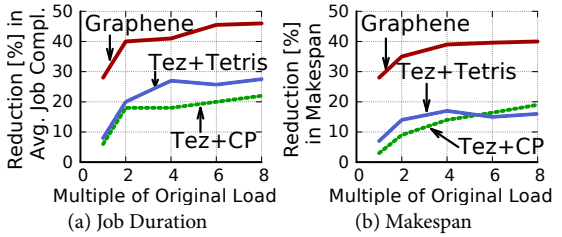
Figure 22: GRAPHENE - sensitivity analysis.

8.4 Sensitivity Analysis

Packing vs. Shortest Remaining Processing Time (srpt): Recall that we combine packing score and srpt using a weighted sum with η (§5). Let η be m times the average over the two expressions that it combines. Figure 22 shows the reduction in average JCT (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 JCT because the effect of srpt reduces; larger values lead to moderately worse makespan. Hence, we recommend $m = 0.2$.

Remote Penalty: GRAPHENE uses a remote penalty rp to prefer local placement. Our analysis shows that both JCT and makespan improve the most when rp $\in [0.7, 0.85]$ (Figure 22). 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. We use rp = 0.8.

Cluster Load: We vary cluster load by reducing the num-



(a) Job Duration (b) Makespan

Figure 23: GRAPHENE's gains increase with cluster load.

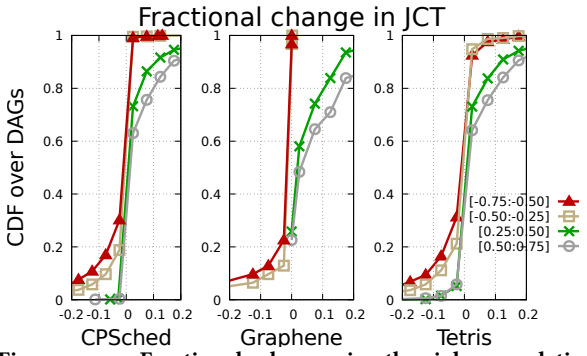
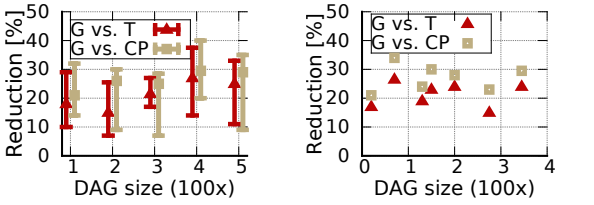


Figure 24: Fractional change in the job completion time (JCT) of DAGs with various schedulers when task durations and resource profiles are mis-estimated.

ber of available servers without changing the workload. Figure 23 shows the JCTs and makespan for a query set derived from TPC-DS. Both GRAPHENE and the alternatives offer more gains at higher loads. This is because the need for careful scheduling and packing increases when resources are scarce. Gains due to GRAPHENE increase by +10% at 2× load and by +15% at 6× load. Further, the gap between GRAPHENE and the alternatives remains similar across load levels.

Impact of misestimations: We offer to each scheduler an inaccurate task duration and resource usage vector but have the underlying execution use the true values. Hence, the schedulers match tasks to machine based on imperfect estimates. Once scheduled, the tasks may finish after a different duration or use different amounts of resources. When the total resource demand crosses machine capacity, we delay the completion of tasks further by a proportional amount. Figure 24 shows a CDF of the change in the completion time of the production DAGs for different schedulers. Each line denotes a different amount of error. For example, the red triangle line labeled $[-0.75 : -0.50]$ corresponds to picking a random number in that range for each stage and then changing the task durations and resource needs fractionally by that random number (-0.75 indicates a 75% lower value). We see that the impact of mis-estimates is rather small; GRAPHENE changes roughly similarly to the other schedulers. Under-estimates tend to speed up the job because the scheduler over-allocates tasks but over-allocation can also slow-down jobs. Over-estimates delay jobs because the scheduler wastes resources; it may refrain from allocating a task when its needs appear larger than the available resources at a machine. Overall, GRAPHENE appears robust to mis-estimations.



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

Figure 25: Comparing GRAPHENE (G) with Tetris (T) and Critical path scheduling (CP) on DAGs from other domains.

9 Applying GRAPHENE to other domains

We evaluate GRAPHENE's effectiveness in scheduling DAGs that arise in distributed compilation jobs [3, 32, 34] and Internet service workflows [46].

Distributed build systems speed up the compilation of large code bases [3, 34]. Each build is a DAG with dependencies between the various tasks (compilation, linking, test, code analysis). The tasks have different runtimes and different resource profiles. Figure 25a shows that GRAPHENE is 20% (30%) faster than `Tetris` (CP) when scheduling the build DAGs from a production distributed build system [32]. Each bar is centered on the median gain for DAGs of a certain size; the error bars are quartiles.

We also examine the DAGs that arise in datacenter-side workflows for Internet-services [46]. 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 run on the same server pool [46]. Over several workflows from a production service, Figure 25b shows that GRAPHENE improves upon alternatives by about 24%. These encouraging early results hint that GRAPHENE may be more broadly useful.

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? GRAPHENE 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, 37, 60, 65, 71]. The best results are when the demand vectors are *small* [21]. Other work considers dependencies but assumes homogeneous demands [29, 36]. 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 Tetris performs poorly when scheduling DAGs (can be up to $2d$ times off, see [38]). Tetris can also be arbitrarily unfair.

Q1 : YES, Q2 : NO. The packing+dependencies problem has been considered at length under *job-shop scheduling* [31, 35, 50, 63]. Most results assume knowledge of job arrival times and profiles [49]. For the case with unknown future job arrivals (the version considered here), no algorithms with bounded competitive ratios are known [54, 55]. Some notable work assumes only two resources [23], applies for a chain but not a general DAG [18] or assumes one cluster-wide resource pool [51].

Q3 : NO. Several of the schemes listed above are complex and hence do not meet the tight timing requirements of cluster schedulers. VM allocators [28] also consider 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 works in cluster scheduling exist such as Quincy [44], Omega [62], Borg [68], Kubernetes [9] and Autopilot [42]. None of these combine multi-resource packing with DAG-awareness and many do neither. Job managers such as Tez [2] and Dryad [43] use simple heuristics such as breath-first scheduling and perform poorly in our experiments.

Q4 : NO. Recently proposed fairness schemes incorporate multiple resources [33] and some are work-conserving [27]. We note that these fairness schemes neither pack nor are DAG-aware. Graphene can incorporate these fairness methods as one of the multiple objectives and trades off bounded unfairness for performance.

11 Concluding Remarks

DAGs are a common scheduling abstraction. However, we found that existing algorithms make key assumptions that do not hold in the case of cluster schedulers. Our scheduler, Graphene, 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 technical contributions are: (1) construct a good schedule for a DAG by placing tasks out-of-order on to a virtual resource-time space, and (2) use an online heuristic to softly enforce the desired schedules and simultaneously manage other concerns such as packing and fairness. 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 Graphene implementation to Apache YARN/Tez projects.

Acknowledgments

For early discussions, we would like to thank Ganesh Ananthanaryanan and Peter Bodik. For feedback that helped improve this paper, we thank the anonymous reviewers, our shepherd Phil Levis, Mohammad Alizadeh, Chris Douglas, Hongzi Mao, Ishai Menache and Malte Schwarzkopf. For operating the cluster that motivated and inspired this work, we thank the Cosmos/ SCOPE production team at Microsoft.

References

[1] 43 bigdata platforms and bigdata analytics software. http://bit.ly/1DROqgt.

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

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

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

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

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

[7] Hadoop MapReduce - Capacity Scheduler. http://bit.ly/1tGpbDN.

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

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

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

[11] Node labels in yarn. http://bit.ly/2d92ook.

[12] Reserved containers in yarn. http://bit.ly/2ckArDO.

[13] Slot and resource fairness in yarn. http://bit.ly/2ckArDO.

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

[15] TPC-DS Benchmark. http://bit.ly/1J6uDap, 2012.

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

[17] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. 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., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. 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).

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

[22] Azar, Y., Kalp-Shaltiel, I., Lucier, B., Menache, I., Naor, J., and Yaniv, J. Truthful online scheduling with commitments. In *EC* (2015).

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

[24] Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB* (2008).

[25] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. Flumejava: Easy, efficient 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, N., Rahman, M., and Boutaba, R. Virtual Network Embedding with Coordinated Node and Link Mapping. In *INFOCOM* (2009).

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

[30] Curino, C., Difallah, D. E., Douglas, C., Krishnan, S., Ramakrishnan, R., and Rao, S. Reservation-based scheduling: If you're late don't blame us! In *SOCC* (2014).

[31] Czumaj, A., and Scheideler, C. A New Algorithm Approach to the General Lovasz Local Lemma with Applications to Scheduling and Satisfiability Problems (Extended Abstract). In *STOC* (2000).

[32] Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., and Kandula, S. CloudBuild: Microsoft's Distributed and Caching Build Service. In *ICSE* (2016).

[33] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. 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] Grandl, R., Kandula, S., Rao, S., Akella, A., and Kulkarni, J. Graphene: Packing and dependency-aware scheduling for data-parallel clusters, extended version. In *MSR Technical Report* (2016).

[39] Greenberg, A., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Vl2: A scalable and flexible data center network. In *SIGCOMM* (2009).

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

[41] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI* (2011).

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

[43] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Eurosys* (2007).

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

[45] 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).

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

[47] Kelly, F., Maulloo, A., and Tan, D. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society* (1998).

[48] Kumar, V. S. A., Marathe, M. V., Parthasarathy, S., and Srinivasan, A. Scheduling on unrelated machines under tree-like precedence constraints. *Algorithmica* (2009).

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

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

[51] 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).

[52] Lucier, B., Menache, I., Naor, J., and Yaniv, J. Efficient online scheduling for deadline-sensitive batch computing. In *SPAA* (2013).

[53] Makarychev, K., and Panigrahi, D. Precedence-constrained scheduling of malleable jobs with preemption. In *ICALP* (2014).

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

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

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

[57] Olston, C., et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD* (2008).

[58] Panigrahy, R., Talwar, K., Uyeda, L., and Wieder, U. Heuristics for Vector Bin Packing. In *MSR TR* (2011).

[59] Peng, D., and Dabek, F. Large-scale incremental processing using distributed transactions and notifications. *DBMS 2006* (2010), 1–15.

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

[61] Schurman, E., and Brutlag, J. The 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., and Varghese, G. Efficient fair queueing using deficit round robin. In *SIGCOMM* (1995).

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

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

[67] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R. Hive- a warehousing solution over a map-reduce framework. In *VLDB* (2009).

[68] 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).

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

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

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

[72] Zaharia, M., Borthakur, D., Sarma, J. S., Elmeleegy, K., Shenker, S., and Stoica, I. Delay Scheduling: A Technique For Achieving Locality And Fairness In Cluster Scheduling. In *EuroSys* (2010).

[73] Zaharia, M., Chowdhury, N. M. M., Franklin, M., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. No. UCB/EECS-2010-53.

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