Firmament: Fast, Centralized Cluster Scheduling at Scale

Ionel Gog, ***University of Cambridge;*** **Malte Schwarzkopf,** ***MIT CSAIL;*** **Adam Gleave and Robert N. M. Watson,** ***University of Cambridge;*** **Steven Hand,** ***Google, Inc.***

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

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Firmament: fast, centralized cluster scheduling at scale

Ionel Gog[†] Malte Schwarzkopf[‡] Adam Gleave[†] Robert N. M. Watson[†] Steven Hand[*]
[†] *University of Cambridge Computer Laboratory* [‡] *MIT CSAIL* [*] *Google, Inc.*

Abstract

Centralized datacenter schedulers can make high-quality placement decisions when scheduling tasks in a cluster. Today, however, high-quality placements come at the cost of high latency at scale, which degrades response time for interactive tasks and reduces cluster utilization.

This paper describes Firmament, a centralized scheduler that scales to over ten thousand machines at sub-second placement latency even though it continuously reschedules all tasks via a min-cost max-flow (MCMF) optimization. Firmament achieves low latency by using multiple MCMF algorithms, by solving the problem incrementally, and via problem-specific optimizations.

Experiments with a Google workload trace from a 12,500-machine cluster show that Firmament improves placement latency by 20× over Quincy [22], a prior centralized scheduler using the same MCMF optimization. Moreover, even though Firmament is centralized, it matches the placement latency of distributed schedulers for workloads of short tasks. Finally, Firmament exceeds the placement quality of four widely-used centralized and distributed schedulers on a real-world cluster, and hence improves batch task response time by 6×.

1 Introduction

Many applications today run on large datacenter clusters [3]. These clusters are shared by applications of many organizations and users [6; 21; 35]. Users execute *jobs*, which each consist of one or more parallel *tasks*. The *cluster scheduler* decides how to place these tasks on cluster machines, where they are instantiated as processes, containers, or VMs.

Better task placements by the cluster scheduler lead to higher machine utilization [35], shorter batch job runtime, improved load balancing, more predictable application performance [12; 36], and increased fault tolerance [32]. Achieving high task *placement quality* is hard: it requires algorithmically complex optimization in multiple dimensions. This goal conflicts with the need for a low *placement latency*, the time it takes the scheduler to place a new task. A low placement latency is required both to meet user expectations and to avoid idle cluster resources while there are waiting tasks. Shorter batch task runtimes and increasing cluster scale make it difficult to meet both conflicting goals [9; 10; 13; 23; 29]. Current schedulers thus choose one to prioritize.

Three different cluster scheduler architectures exist today. First, *centralized* schedulers use elaborate algorithms to find high-quality placements [11; 12; 35], but have latencies of seconds or minutes [13; 32]. Second, *distributed* schedulers use simple algorithms that allow for high throughput, low latency parallel task placement at scale [13; 28; 29]. However, their uncoordinated decisions based on partial, stale state can result in poor placements. Third, *hybrid* schedulers split the workload across a centralized and a distributed component. They use sophisticated algorithms for long-running tasks, but rely on distributed placement for short tasks [9; 10; 23].

In this paper, we show that a centralized scheduler based on sophisticated algorithms can be fast and scalable for both current and future workloads. We built Firmament, a centralized scheduler that meets three goals:

1. to maintain the same high placement quality as an existing centralized scheduler (*viz.* Quincy [22]);
2. to achieve sub-second task placement latency for all workloads in the common case; and
3. to cope well with demanding situations such as cluster oversubscription or large incoming jobs.

Our key insight is that even centralized sophisticated algorithms for the scheduling problem can be fast (*i*) if they match the problem structure well, and (*ii*) if few changes to cluster state occur while the algorithm runs.

Firmament generalizes Quincy [22], which represents the scheduling problem as a min-cost max-flow (MCMF) optimization over a graph (§3) and continuously reschedules the entire workload. Quincy's original MCMF algorithm results in task placement latencies of minutes on a large cluster. Firmament, however, achieves placement

latencies of hundreds of milliseconds in the common case and reaches the same placement quality as Quincy.

To achieve this, we studied several MCMF optimization algorithms and their performance (§4). Surprisingly, we found that *relaxation* [4], a seemingly inefficient MCMF algorithm, outperforms other algorithms on the graphs generated by the scheduling problem. However, relaxation can be slow in crucial edge cases, and we thus investigated three techniques to reduce Firmament's placement latency across different algorithms (§5):

1. Terminating the MCMF algorithms early to find *approximate solutions* generates unacceptably poor and volatile placements, and we reject the idea.
2. *Incremental re-optimization* improves the runtime of Quincy's original MCMF algorithm (cost scaling [17]), and makes it an acceptable fallback.
3. *Problem-specific heuristics* aid some MCMF algorithms to run faster on graphs of specific structure.

We combined these algorithmic insights with several implementation-level techniques to further reduce Firmament's placement latency (§6). Firmament runs two MCMF algorithms concurrently to avoid slowdown in edge cases; it implements an efficient graph update algorithm to handle cluster state changes; and it quickly extracts task placements from the computed optimal flow.

Our evaluation compares Firmament to existing distributed and centralized schedulers, both in simulation (using a Google workload trace) and on a local 40-machine cluster (§7). In our experiments, we find that Firmament scales well: even with 12,500 machines and 150,000 live tasks eligible for rescheduling, Firmament makes sub-second placements. This task placement latency is comparable to those of distributed schedulers, even though Firmament is centralized. When scheduling workloads that consist *exclusively* of short, sub-second tasks, Firmament scales to over 1,000 machines, but suffers overheads for task runtimes below 5s at 10,000 machines. Yet, we find that Firmament copes well with realistic, mixed workloads that combine long-running services and short tasks even at this scale: Firmament keeps up with a 250× accelerated Google workload. Finally, we show that Firmament's improved placement quality reduces short batch tasks' runtime by up to 6× compared to other schedulers on a real-world cluster.

Firmament is available as open-source software (§9).

2 Background

Cluster managers such as Mesos [21], YARN [34], Borg [35], and Kubernetes [14] automatically share and manage physical datacenter resources. Each one has a *scheduler*, which is responsible for placing tasks on machines. Figure 1 illustrates the lifecycle of a task in a cluster manager: after the user submits the task, it waits until the scheduler places it on a machine where it subsequently runs. The time between submission and task placement is the *task placement latency*, and to the total time between the task's submission and its completion is the *task response time*.[1] The time a task spends being actively scheduled is the scheduler's *algorithm runtime*.

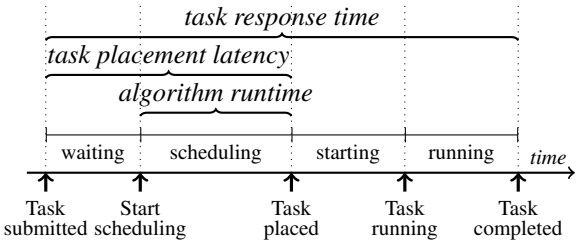


Figure 1: Task lifecycle phases, state transition events *(bottom)* and the time ranges used in this paper *(top)*.

For each task, the scheduling algorithm typically first performs a *feasibility check* to identify suitable machines, then *scores* them according to a preference order, and finally *places* the task on the best-scoring machine. Scoring, i.e., rating the different placement choices for a task, can be expensive. Google's Borg, for example, relies on several batching, caching, and approximation optimizations to keep scoring tractable [35, §3.4].

High placement quality increases cluster utilization and avoids performance degradation due to overcommit. Poor placement quality, by contrast, increases task response time (for batch tasks), or decreases application-level performance (for long-running services).

2.1 Task-by-task placement

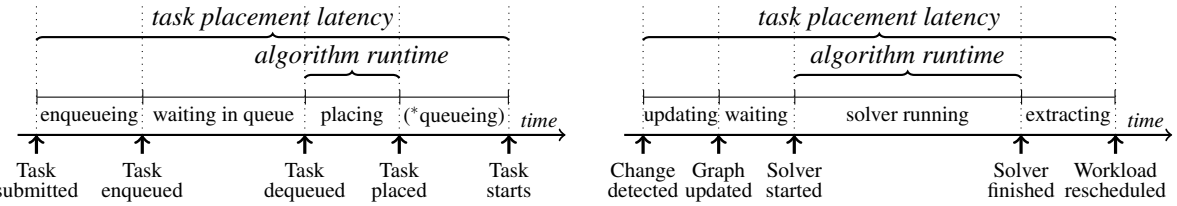
Most cluster schedulers, whether centralized or distributed, are *queue-based* and process one task at a time (per scheduler). Figure 2a illustrates how such a queue-based scheduler processes a new task. The task first waits in a queue of unscheduled tasks until it is dequeued and processed by the scheduler. In a busy cluster, a task may spend substantial time enqueued. Some schedulers also have tasks wait in a per-machine "worker-side" queue [29], which allows for pipelined parallelism.

Task-by-task placement has the advantage of being amenable to uncoordinated, parallel decisions in distributed schedulers [9; 10; 13; 28]. On the other hand, processing one task at a time also has two crucial downsides: first, the scheduler commits to a placement early and restricts its choices for further waiting tasks, and second, there is limited opportunity to amortize work.

2.2 Batching placement

Both downsides of task-by-task placement can be addressed by batching. Processing several tasks in a batch

[1] Task response time is primarily meaningful for batch tasks; long-running service tasks' response times are conceptually infinite, and in practice are determined by failures and operational decisions.



(a) Queue-based schedulers (*: optional worker-side queue). **(b)** Flow-based schedulers (Quincy [22], Firmament).

Figure 2: Tasks wait to be placed individually in queue-based schedulers (a), while flow-based schedulers (b) reschedule the whole workload in a long solver run, which makes it essential to minimize algorithm runtime at scale.

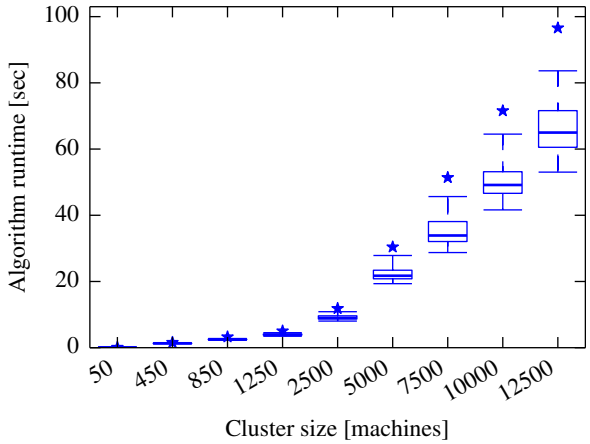


Figure 3: Quincy [22]'s approach scales poorly as cluster size grows. Simulation on subsets of the Google trace; boxes are 25th, 50th, and 75th percentile delays, whiskers 1st and 99th, and a star indicates the maximum.

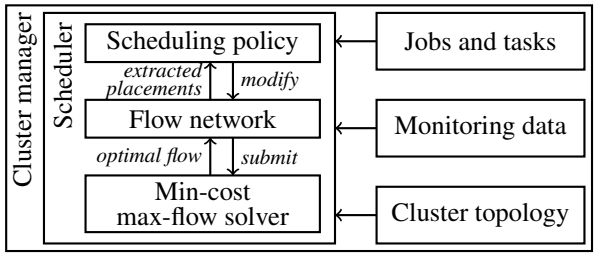


Figure 4: Firmament's scheduling policy modifies the flow network according to workload, cluster, and monitoring data; the network is passed to the MCMF solver, whose computed optimal flow yields task placements.

allows the scheduler to jointly consider their placement, and thus to find the best trade-off for the whole batch. A natural extension of this idea is to reconsider the *entire* existing workload ("rescheduling"), and to preempt and migrate running tasks if prudent.

Flow-based scheduling, introduced by Quincy [22], is an efficient batching technique. Flow-based scheduling uses a placement mechanism – min-cost max-flow (MCMF) optimization – with an attractive property: it guarantees overall optimal task placements for a given scheduling policy. Figure 2b illustrates how it proceeds. If a change to cluster state happens (e.g., task submission), the scheduler updates an internal graph representation of the scheduling problem. It waits for any running optimization to finish, and then runs a MCMF solver on the graph. This yields an optimal flow from which the scheduler extracts the task assignments.

However, Figure 3 illustrates that Quincy, the current state-of-the-art flow-based scheduler, is too slow to meet our placement latency goal at scale. In this experiment, we replayed subsets of the public Google trace [30], which we augmented with locality preferences for batch processing jobs[2] against our faithful reimplementation of Quincy's approach. We measured the scheduler algorithm runtime for clusters of increasing size with proportional workload growth. The algorithm runtime increases with scale, up to a median of 64s and a 99th percentile of 83s for the full Google cluster (12,500 machines). During this time, the scheduler must wait for the solver to finish, and cannot choose any placements for new tasks.

The goal of this paper is to build a flow-based scheduler that achieves equal placement quality to Quincy, but which does so at sub-second placement latency. As our experiment illustrates, we must achieve at least an order-of-magnitude speedup over Quincy to meet this goal.

3 Firmament approach

We chose to develop Firmament as a flow-based scheduler for three reasons. First, flow-based scheduling considers the entire workload, allowing us to support rescheduling and priority preemption. Second, flow-based scheduling achieves high placement quality and, consequently, low task response times [22, §6]. Third, as a batching approach, flow-based scheduling amortizes work well over many tasks and placement decisions, and hence achieves high task throughput – albeit at a high placement latency that we aim to improve.

[2]Details of our simulation are in §7; in the steady-state, the 12,500-machine cluster runs about 150,000 tasks comprising about 1,800 jobs.

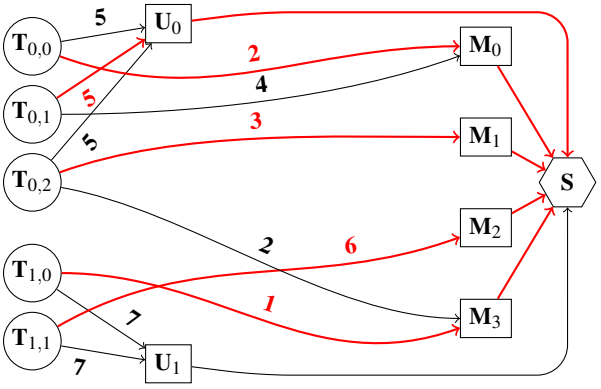


Figure 5: Example flow network for a four-machine cluster with two jobs of three and two tasks. All tasks except $\mathbf{T}_{0,1}$ are scheduled on machines. Arc labels show non-zero costs, and all arcs have unit capacity apart from those between unscheduled aggregators and the sink. The red arcs carry flow and form the min-cost solution.

3.1 Architecture

Figure 4 gives an overview of the Firmament scheduler architecture. Firmament, like Quincy, models the scheduling problem as a min-cost max-flow (MCMF) optimization over a *flow network*. The flow network is a directed graph whose structure is defined by the *scheduling policy*. In response to events and monitoring information, the flow network is modified according to the scheduling policy, and submitted to an *MCMF solver* to find an optimal (i.e., min-cost) flow. Once the solver completes, it returns the optimal flow, from which Firmament extracts the implied task placements. In the following, we first explain the basic structure of the flow network, and then discuss how to make the solver fast.

3.2 Flow network structure

A flow network is a directed graph whose arcs carry *flow* from source nodes to a sink node. A *cost* and *capacity* associated with each arc constrain the flow, and specify preferential routes for it.

Figure 5 shows an example of a flow network that expresses a simple cluster scheduling problem. Each task node $\mathbf{T}_{j,i}$ on the left hand side, representing the i^{th} task of job j, is a source of one unit of flow. All such flow must be drained into the sink node ($\mathbf{S}$) for a feasible solution to the optimization problem. To reach $\mathbf{S}$, flow from $\mathbf{T}_{j,i}$ can proceed through a machine node ($\mathbf{M}_m$), which schedules the task on machine m (e.g., $\mathbf{T}_{0,2}$ on $\mathbf{M}_1$). Alternatively, the flow may proceed to the sink through an unscheduled aggregator node ($\mathbf{U}_j$ for job j), which leaves the task unscheduled (as with $\mathbf{T}_{0,1}$) or preempts it if running.

In the example, a task's placement preferences are expressed as costs on direct arcs to machines. The cost to leave the task unscheduled, or to preempt it when running, is the cost on its arc to the unscheduled aggregator (e.g., 7 for $\mathbf{T}_{1,1}$). Given this flow network, an MCMF solver finds a globally optimal (i.e., minimum-cost) flow (shown in red in Figure 5). This optimal flow expresses the best trade-off between the tasks' unscheduled costs and their placement preferences. Task placements are extracted by tracing flow from the machines back to tasks.

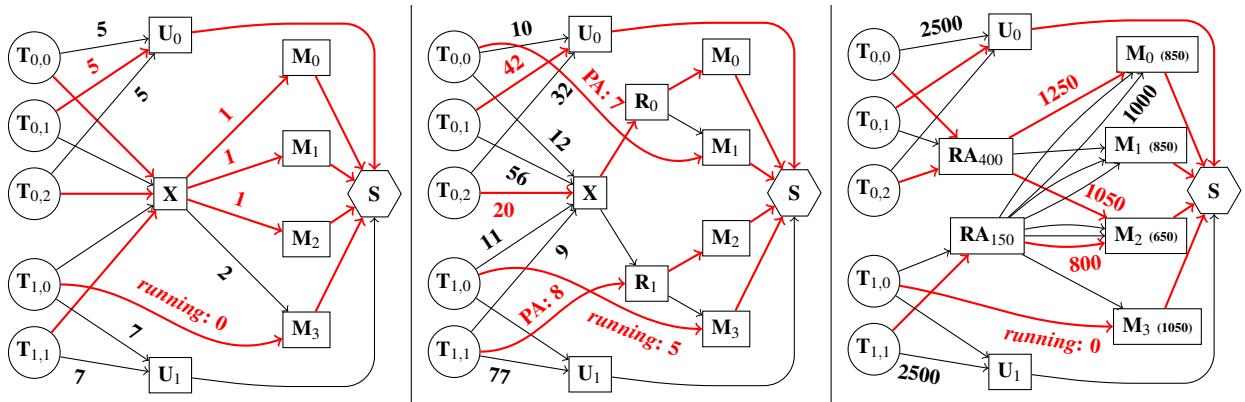
In our example, tasks had only direct arcs to machines. The solver finds the best solution if every task has an arc to each machine scored according to the scheduling policy, but this requires thousands of arcs per task on a large cluster. Policy-defined *aggregator* nodes, similar to the unscheduled aggregators, reduce the number of arcs required to express a scheduling policy. Such aggregators group, e.g., machines in a rack, tasks with similar resource needs, or machines with similar capabilities. With aggregators, the cost of a task placement is the sum of all costs on the path from the task node to the sink.

3.3 Scheduling policies

Firmament generalizes flow-based scheduling over the single, batch-oriented policy proposed by Quincy. Cluster administrators use a policy API to configure Firmament's scheduling policy, which may incorporate e.g., multi-dimensional resources, fairness, and priority preemption [31, Ch. 6–7]. This paper focuses on Firmament's scalability, and we therefore use only three simplified, illustrative policies explained in the following: (*i*) a simple load-spreading policy, (*ii*) Quincy's slot-based, locality-oriented policy, and (*iii*) a network-aware policy that avoids overloading machines' network connections.

Load-spreading policy. Figure 6a shows a trivial use of an aggregator: all tasks have arcs to a cluster-wide aggregator ($\mathbf{X}$). The cost on the outgoing arc from $\mathbf{X}$ to each machine node is proportional to the number of tasks already running on the machine (e.g., one task on $\mathbf{M}_3$). The effect is that the number of tasks on a machine only increases once all other machines have at least as many tasks (as e.g., in Docker SwarmKit). This policy neither requires or nor uses the full sophistication of flow-based scheduling. We use it to highlight specific edge cases in MCMF algorithms (see §4.3).

Quincy policy. Figure 6b depicts Quincy's original locality-oriented policy [22, §4.2], which uses rack aggregators ($\mathbf{R}_r$) and a cluster aggregator ($\mathbf{X}$) to express data locality for batch jobs. Tasks have low-cost *preference arcs* to machines and racks on which they have local data, but fall back to scheduling via the cluster aggregator if their preferences are unavailable (e.g., $\mathbf{T}_{0,2}$). This policy is suitable for batch jobs, and optimizes for a trade-off between data locality, task wait time, and preemption cost. We use it to illustrate MCMF algorithm performance and for head-to-head comparison with Quincy.



(a) Load-spreading policy with a single cluster aggregator (**X**) and costs proprtional to number of tasks per machine.

(b) Quincy policy with cluster (**X**) and rack (**R**) aggregators, and data locality preference arcs (PA).

(c) Network-aware policy with request aggregators (**RA**) and dynamic arcs to machines with spare network bandwidth.

Figure 6: Different aggregators, arcs, and costs help Firmament express the scheduling policies used in this paper; costs are example values consistent with each policy. Firmament also supports other policies via an API [31, Ch. 6–7].

Network-aware policy. Figure 6c illustrates a policy which avoids overcommitting machines' network bandwidth (which degrades task response time). Each task connects to a request aggregator (**RA**) for its network bandwidth request. The **RA**s have one arc for each task that fits on each machine with sufficient spare bandwidth (e.g., 650 MB/s of 1.25 GB/s on $\mathbf{M}_2$'s 10G link). These arcs are dynamically adapted as the observed bandwidth use changes. Costs on the arcs to machines are the sum of the request and the currently used bandwidth, which incentivizes balanced utilization. We use this policy to illustrate Firmament's potential to make high-quality decisions, but a production policy would be more complex and extend it with a priority notion and additional resource dimensions (e.g., CPU/RAM) [31, §7.3].

4 Min-cost max-flow algorithms

A flow-based scheduler can use any MCMF algorithm, but some algorithms are better suited to the scheduling problem than others. In this section, we explain the MCMF algorithms that we implemented for Firmament, compare them empirically, and explain their sometimes unexpected performance.

A min-cost max-flow algorithm takes a directed flow network $G = (N, A)$ as input. Each arc $(i, j) \in A$ has a cost c_{ij} and a maximum capacity u_{ij}. Each node $i \in N$ also has an associated supply $b(i)$; nodes with positive supply are *sources*, those with negative supply are *sinks*.

Informally, MCMF algorithms must optimally route the flow from all sources (e.g., task nodes $\mathbf{T}_{i,j}$) to sinks (e.g., the sink node **S**) without exceeding the capacity constraint on any arc. To understand the differences between MCMF algorithms, we need a slightly more formal definition: the goal is to find a flow f that minimizes Eq. 1, while respecting the flow *feasibility constraints* of **mass balance** (Eq. 2) and **capacity** (Eq. 3):

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij} f_{ij} \text{ subject to} \tag{1}$$

$$\sum_{k:(j,k)\in A} f_{jk} - \sum_{i:(i,j)\in A} f_{ij} = b(j), \forall j \in N \tag{2}$$

$$\text{and } 0 \leq f_{ij} \leq u_{ij}, \forall (i,j) \in A \tag{3}$$

Some algorithms use an equivalent definition of the flow network, the *residual network*. In the residual network, each arc $(i, j) \in A$ with cost c_{ij} and maximum capacity u_{ij} is replaced by two arcs: (i, j) and (j, i). Arc (i, j) has cost c_{ij} and a *residual capacity* of $r_{ij} = u_{ij} - f_{ij}$, while arc (j, i) has cost $-c_{ij}$ and a residual capacity $r_{ji} = f_{ij}$. The feasibility constraints also apply in the residual network.

The *primal* minimization problem (Eq. 1) also has an associated *dual* problem, which some algorithms solve more efficiently. In the dual min-cost max-flow problem, each node $i \in N$ has an associated dual variable $\pi(i)$ called the *potential*. The potentials are adjusted in different, algorithm-specific ways to meet optimality conditions. Moreover, each arc has a *reduced cost* with respect to the node potentials, defined as:

$$c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j) \tag{4}$$

A feasible flow is optimal if and only if at least one of three *optimality conditions* is met:

1. **Negative cycle optimality**: no directed negative-cost cycles exist in the residual network.
2. **Reduced cost optimality**: there is a set of node potentials π such that there are no arcs in the residual network with negative reduced cost (c_{ij}^{π}).

Algorithm	Worst-case complexity
Relaxation	$O(M^3CU^2)$
Cycle canceling	$O(NM^2CU)$
Cost scaling	$O(N^2M\log(NC))$
Successive shortest path	$O(N^2U\log(N))$

Table 1: Worst-case time complexities for min-cost max-flow algorithms. N is the number of nodes, M the number of arcs, C the largest arc cost and U the largest arc capacity. In our problem, $M > N > C > U$.

3. **Complementary slackness optimality**: there is a set of node potentials π such that the flow on arcs with $c_{ij}^{\pi} > 0$ is zero, and there are no arcs with both $c_{ij}^{\pi} < 0$ and available capacity.

Algorithms. The simplest MCMF algorithm is **cycle canceling** [25]. The algorithm first computes a max-flow solution, and then performs a series of iterations in which it augments flow along negative-cost directed cycles in the residual network. Pushing flow along such a cycle guarantees that the overall solution cost decreases. The algorithm finishes with an optimal solution once no negative-cost cycles remain (i.e., the negative cycle optimality condition is met). Cycle canceling always maintains feasibility and attempts to achieve optimality.

Unlike cycle canceling, the **successive shortest path** algorithm [2, p. 320] maintains reduced cost optimality at every step and tries to achieve feasibility. It repeatedly selects a source node (i.e., $b(i) > 0$) and sends flow from it to the sink along the shortest path.

The **relaxation** algorithm [4; 5], like successive shortest path, augments flow from source nodes along the shortest path to the sink. However, unlike successive shortest path, relaxation optimizes the dual problem by applying one of two changes when possible:

1. Keeping π unchanged, the algorithm modifies the flow, f, to f' such that f' still respects the reduced cost optimality condition and the total supply decreases (i.e., feasibility improves).
2. It modifies π to π' and f to f' such that f' is still a reduced cost-optimal solution and the cost of that solution decreases (i.e., total cost decreases).

This allows relaxation to decouple the improvements in feasibility from reductions in total cost. When relaxation can reduce cost or improve feasibility, it reduces cost.

Cost scaling [17–19] iterates to reduce cost while maintaining feasibility, and uses a relaxed complementary slackness condition called ε-optimality. A flow is ε-optimal if the flow on arcs with $c_{ij}^{\pi} > \varepsilon$ is zero and there are no arcs with $c_{ij}^{\pi} < -\varepsilon$ on which flow can be sent. Initially, ε is equal to the maximum arc cost, but ε rapidly decreases as it is divided by a constant factor after every iteration that achieves ε-optimality. Cost scaling finishes when $\frac{1}{n}$-optimality is achieved, since this is equivalent to the complementary slackness optimality condition [17].

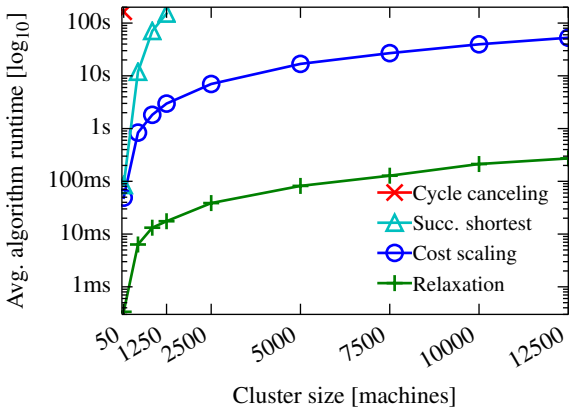


Figure 7: Average runtime for MCMF algorithms on clusters of different sizes, subsampled from the Google trace. We use the Quincy policy and slot utilization is about 50%. Relaxation performs best, despite having the highest time complexity. [*N.B.:* $\log_{10}$-scale y-axis.]

4.1 Algorithmic performance

Table 1 summarizes the worst-case complexities of the algorithms discussed. The complexities suggest that successive shortest path ought to work best, as long as $U\log(N) < M\log(NC)$, which is the case as $U \ll M$ and $C \geq 1$. However, since MCMF algorithms are known to have variable runtimes depending on the input graph [15; 24; 26], we decided to directly measure performance.

4.2 Measured performance

As in the experiment in Figure 3, we subsample the Google trace and replay it for simulated clusters of different sizes. We use the Quincy scheduling policy for batch jobs and prioritize service jobs over batch ones. Figure 7 shows the average runtime for each MCMF algorithm considered. Even though it has the best worst-case time complexity, successive shortest path outperforms only cycle canceling, and even on a modest cluster of 1,250 machines its algorithm runtime exceeds 100 seconds.

Moreover, the relaxation algorithm, which has the highest worst-case time complexity, actually performs best in practice. It outperforms cost scaling (used in Quincy) by two orders of magnitude: on average, relaxation completes in under 200ms even on a cluster of 12,500 machines. One key reason for this perhaps surprising performance is that relaxation does minimal work when most scheduling choices are straightforward. This happens if the destinations for tasks' flow are uncontested, i.e., few new tasks have arcs to the same location and attempt to schedule there. In this situation, relaxation routes most of the flow in a single pass over the graph.

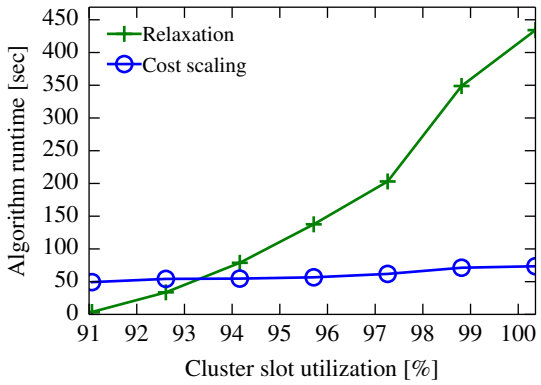


Figure 8: Close to full cluster utilization, relaxation runtime increases dramatically, while cost scaling is unaffected: the *x*-axis shows the utilization after scheduling jobs of increasing size to a 90%-utilized cluster.

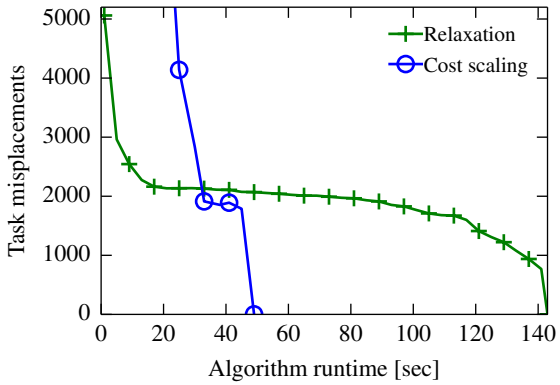


Figure 10: Approximate min-cost max-flow yields poor solutions, since many tasks are misplaced until shortly before the algorithms reach the optimal solution.

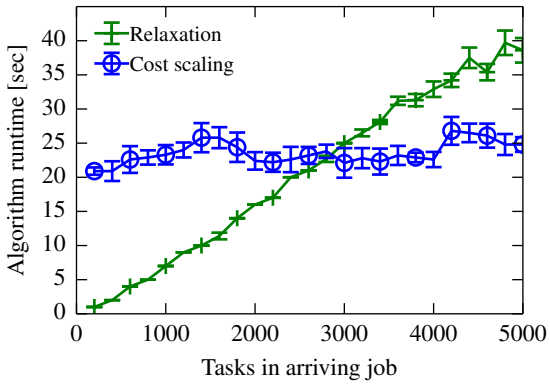


Figure 9: Contention slows down the relaxation algorithm: on cluster with a load-spreading scheduling policy, relaxation runtime exceeds that of cost scaling at just under 3,000 concurrently arriving tasks (e.g., a large job).

4.3 Edge cases for relaxation

Yet, relaxation is not always the right choice. For example, it can perform poorly under the high load and oversubscription common in batch analytics clusters [29]. Figure 8 illustrates this: here, we push the simulated Google cluster closer to oversubscription. We take a snapshot of the cluster and then submit increasingly larger jobs. The relaxation runtime increases rapidly, and at about 93% cluster utilization, it exceeds that of cost scaling, growing to over 400s in the oversubscribed case.

Moreover, some scheduling policies *inherently* create contention between tasks. Consider, for example, our load-spreading policy that balances the task count on each machine. This policy makes "under-populated" machines a popular destination for tasks' flow, and thus creates contention. We illustrate this with an experiment: we submit a single job with an increasing number of tasks to a cluster using the load-spreading policy. This corresponds to the rare-but-important arrival of very large jobs: for example, 1.2% of jobs in the Google trace have over 1,000 tasks, and some even over 20,000. Figure 9 shows that relaxation's runtime increases linearly in the number of tasks, and that it exceeds the runtime of cost scaling once the new job has over 3,000 tasks.

To make matters worse, a single overlong relaxation run can have a devastating effect on long-term placement latency. If many new tasks arrive during such a long run, the scheduler might again be faced with many unscheduled tasks when it next runs. Hence, relaxation may take a long time again, accumulate many changes, and in the worst case fail to ever recover to low placement latency.

5 MCMF optimizations for scheduling

Relaxation has promising common-case performance at scale for typical workloads. However, its edge-case behavior makes it necessary either (*i*) to fall back to other algorithms in these cases, or (*ii*) to reduce runtime in other ways. In the following, we use challenging graphs to investigate optimizations that either improve relaxation or the best "fallback" algorithm, cost scaling.

5.1 Approximate min-cost max-flow

MCMF algorithms return an *optimal* solution. For the cluster scheduling problem, however, an approximate solution may well suffice. For example, TetriSched [33] (based on an MILP solver), as well as Paragon [11] and Quasar [12] (based on collaborative filtering), terminate their solution search after a set time. We therefore investigated the solution quality of cost scaling and relaxation when they are terminated early. This would work well if the algorithms spent a long time on minor solution refinements with little impact on the overall outcome.

Algorithm	*Feasiblity*	*Red. cost optimality*	*ε-optimality*
Relaxation	–	✓	–
Cycle canceling	✓	–	–
Cost scaling	✓	–	✓
Succ. shortest path	–	✓	–

Table 2: Algorithms have different preconditions for each internal iteration. Cost scaling expects feasibility and ε-optimality, making it difficult to incrementalize.

In our experiment, we use a highly-utilized cluster (cf. Figure 8) to investigate relaxation and cost scaling, but the results generalize. Figure 10 shows the number of "misplaced" tasks as a function of how early we terminate the algorithms. We treat any task as misplaced if it is (*i*) preempted in the approximate solution but keeps running in the optimal one; (*ii*) scheduled on a different machine to where it is scheduled in the optimal solution. Both cost scaling and relaxation misplace thousands of tasks when terminated early, and tasks are still misplaced even in the final iteration before completion. Hence, early termination appears not to be a viable placement latency optimization for flow-based schedulers.

5.2 Incremental min-cost max-flow

Since cluster state does not change dramatically between subsequent scheduling runs, the MCMF algorithm might be able to reuse its previous state. In this section, we describe what changes are required to make MCMF algorithms work incrementally, and provide some intuition for which algorithms are suitable for incremental use.

All cluster events (e.g., task submissions, machine failures) ultimately reduce to three different types of *graph change* in the flow network:

1. **Supply** changes at nodes when arcs or nodes which previously carried flow are removed (e.g., due to machine failure), or when nodes with supply are added to the graph (e.g., at task submission).
2. **Capacity** changes on arcs if machines fail or (re)join the cluster. Note that arc additions and removals can also be modeled as capacity changes from and to zero-capacity arcs.
3. **Cost** changes on an arc when the desirability of routing flow via that arc changes; when these happen exactly depends on the scheduling policy.

Changes to the supply of a node, an arc's capacity, or its cost can invalidate the feasibility and optimality of an existing flow. Some MCMF algorithms require the flow to be feasible at every step and improve ε-optimality, while others require optimality to always hold and improve feasibility (Table 2). A solution must be optimal *and* feasible because an infeasible solution fails to route all flow, which leaves tasks unscheduled or erroneously preempts them, while a non-optimal solution misplaces tasks.

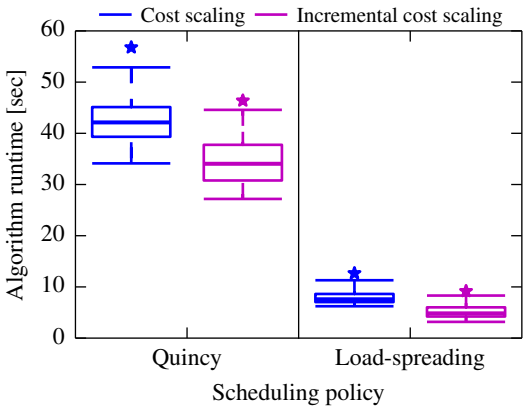


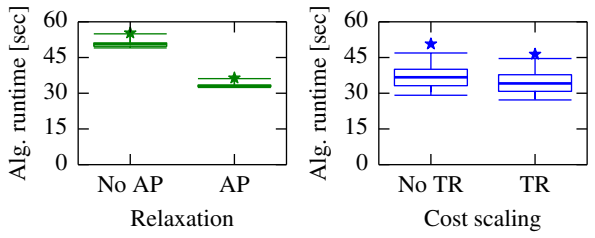
Figure 11: Incremental cost scaling is 25% faster compared to from-scratch cost scaling for the Quincy policy and 50% faster for the load-spreading policy.

	Reduced cost on arc from i to j		
Change type	$c^\pi_{ij} < 0$	$c^\pi_{ij} = 0$	$c^\pi_{ij} > 0$
Increasing arc cap.			
Decreasing arc cap.		$f_{ij} > u_{ij}$	
Increasing arc cost	$c_{ij}' > 0$	$f_{ij} > 0$	
Decreasing arc cost			$c_{ij}' < 0$

Table 3: Arc changes requiring solution reoptimization. *Green:* stays optimal and feasible; *red:* breaks feasibility or optimality; *orange:* breaks feasibility or optimality if condition in cell holds. Decreasing arc capacity can destroy feasibility; all other changes affect optimality only.

We implemented incremental versions of the cost scaling and relaxation algorithms. **Incremental cost scaling** is up to 50% faster than running cost scaling from scratch (Figure 11). Incremental cost scaling's potential gains are limited because cost scaling requires the flow to be feasible and ε-optimal before each intermediate iteration (Table 2). Graph changes can cause the flow to violate one or both requirements: for example, any addition or removal of task nodes adds supply and breaks feasibility. Table 3 shows the effect of different arc changes on the feasibility and optimality of the flow. A change that modifies the cost of an arc (i, j) from $c^\pi_{ij} < 0$ to $c_{ij}' > 0$, for example, breaks optimality. Many changes break optimality and cause cost scaling to fall back to a higher ε-optimality to compensate. To bring ε back down, cost scaling must do a substantial part of the work it would do from scratch. However, the limited improvement still helps reduce our fallback algorithm's runtime.

Incremental relaxation ought to work better than incremental cost scaling, since the relaxation algorithm only needs to maintain reduced cost optimality (Table 2). In practice, however, it turns out not to work well. While the algorithm can be incrementalized with relative ease and often runs faster, it – counter-intuitively – can also be



(a) Arc prioritization (AP). **(b)** Eff. task removal (TR).

Figure 12: Problem-specific heuristics reduce runtime by 45% (AP, relaxation) and 10% (TR, inc. cost scaling).

slower incrementally than when running from scratch.

Relaxation requires reduced cost optimality to hold at every step of the algorithm and tries to achieve feasibility by pushing flow on zero-reduced cost arcs from source nodes to nodes with demand. The algorithm builds a tree of zero-reduced cost arcs from each source node in order to find such demand nodes. The tree is expanded by adding zero-reduced cost arcs to it. When running from scratch, the likelihood of zero-reduced cost arcs connecting two zero-reduced cost trees is low, as there are few such trees initially. Only when the solution is close to optimality, trees are joined into larger ones. Incremental relaxation, however, works with the existing, close-to-optimal state, which already contains large trees that must be extended for each source. Having to traverse these large trees many times, incremental relaxation can run slower than from scratch. This happens especially for graphs that relaxation already struggles with, e.g. ones that contain nodes with a lot of potential incoming flow. In practice, we found that incremental relaxation performs well only if tasks are not typically connected to a large zero-reduced cost tree.

5.3 Problem-specific heuristics

Our scheduler runs min-cost max-flow on a graph with specific properties, rather than the more general graphs typically used to evaluate MCMF algorithms [24, §4]. For example, our graph has a single sink; it is a directed acyclic graph; and flow must always traverse unscheduled aggregators or machine nodes. Hence, problem-specific heuristics might help the algorithms find solutions more quickly. We investigated several such heuristics, and found two beneficial ones: (*i*) prioritization of promising arcs, and (*ii*) efficient task node removal.

5.3.1 Arc prioritization

The relaxation algorithm builds a tree of zero-reduced cost arcs for every source node (see §5.2) in order to locate zero-reduced cost paths (i.e., paths that do not break reduced cost optimality) to nodes with demand. When this tree must be extended, any arc of zero reduced cost that connects a node inside the tree to a node outside the tree can be used. However, some arcs are better choices for extension than others. The quicker we can find paths to nodes with demand, the sooner we can route the supply. We therefore prioritize arcs that lead to nodes with demand when extending the cut, adding them to the front of a priority queue to ensure they are visited sooner.

In effect, this heuristic implements a hybrid graph traversal that biases towards depth-first exploration when demand nodes can be reached, but uses breadth first exploration otherwise. Figure 12a shows that applying this heuristic reduces relaxation runtime by 45% when running over a graph with contended nodes.

5.3.2 Efficient task removal

Our second heuristic helps incremental cost scaling. It is based on the insight that removal of a running task is common (e.g., due to completion, preemption, or a machine failure), but breaks feasibility. This happens because the task node is removed, which creates demand at the machine node where the task ran, since the machine node still has outgoing flow in the intermediate solution. Breaking feasibility is expensive for cost scaling (§5.2).

However, we can reconstruct the task's flow through the graph, remove it, and drain the machine node's flow at the single sink node. This creates demand in a single place only (the sink), which accelerates the incremental solution. However, Figure 12b shows that this heuristic offers only modest gains: it improves runtime by 10%.

6 Firmament implementation

We implemented a new MCMF solver for Firmament. It supports the four algorithms discussed earlier (§4) and incremental cost scaling. The solver consists of about 8,000 lines of C++. Firmament's cluster manager and our simulator are implemented in about 24,000 lines of C++, and are available at `http://firmament.io`.

In this section, we discuss implementation-level techniques that, in addition to our prior algorithmic insights, help Firmament achieve low task placement latency.

6.1 Algorithm choice

In §4, we saw that the practical performance of MCMF algorithms varies. Relaxation often works best, but scales poorly in specific edge cases. Cost scaling, by contrast, scales well and can be incrementalized (§5.2), but is usually substantially slower than relaxation.

Firmament's MCMF solver always speculatively executes cost scaling and relaxation, and picks the solution offered by whichever algorithm finishes first. In the common case, this is relaxation; having cost scaling as well guarantees that Firmament's placement latency does not grow unreasonably large in challenging situations. We run both algorithms instead of developing a heuristic to choose the right one for two reasons: first, it is cheap, as

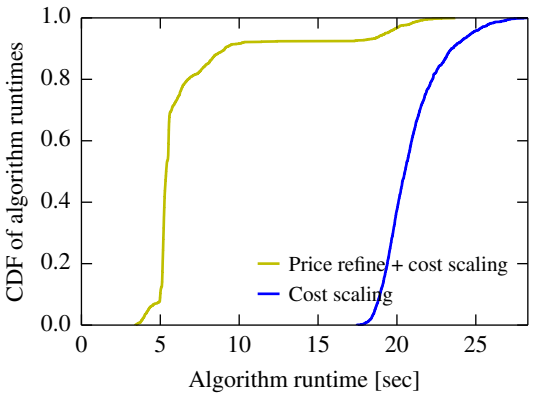


Figure 13: Incremental cost scaling runs 4× faster if we apply the price refine heuristic to a graph from relaxation.

```
to_visit = machine_nodes # list of machine nodes
node_flow_destinations = {} # auxiliary remember set
mappings = {} # final task mappings
while not to_visit.empty():
  node = to_visit.pop()
  if node.type is not TASK_NODE:
    # Visit the incoming arcs
    for arc in node.incoming_arcs():
      moved_machines = 0
      # Move as many machines to the incoming arc's
      # source node as there is flow on the arc
      while assigned_machines < arc.flow:
        node_flow_destinations[arc.source].append(
          node_flow_destinations[node].pop())
        moved_machines += 1
      # (Re)visit the incoming arc's source node
      if arc.source not in to_visit:
        to_visit.append(arc.source)
  else: # node.type is TASK_NODE
    mappings[node.task_id] =
      node_flow_destinations[node].pop()
return mappings
```

Listing 1: Our efficient algorithm for extracting task placements from the optimal flow returned by the solver.

the algorithms are single-threaded and do not parallelize; second, predicting the right algorithm is hard and the heuristic would depend on both scheduling policy and cluster utilization (cf. §4), making it brittle and complex.

6.2 Efficient algorithm switching

Firmament also applies an optimization that helps it efficiently transition state from relaxation to incremental cost scaling. Firmament's MCMF solver uses incremental cost scaling as it is faster than running cost scaling from scratch (§5.2). Typically, however, the (from-scratch) relaxation algorithm finishes first. The next incremental cost scaling run must therefore use the solution from a prior *relaxation* as a starting point. Since relaxation and cost scaling use different reduced cost graph representations, this can be slow. Specifically, relaxation may converge on node potentials that fit poorly into cost scaling's complementary slackness requirement, since relaxation only requires reduced cost optimality.

We found that *price refine* [17], a heuristic originally developed for use within cost scaling, helps with this transition. Price refine reduces the node potentials without affecting solution optimality, and thus simplifies the problem for cost scaling. Figure 13 shows that applying price refine to the prior relaxation solution graph speeds up incremental cost scaling by 4× in 90% of cases.

We apply price refine on the previous solution *before* we apply the latest cluster changes. This guarantees that price refine is able to find node potentials that satisfy complementary slackness optimality without modifying the flow. Consequently, cost scaling must start only at a value of ε equal to the costliest arc graph change.

6.3 Efficient solver interaction

So far, we have primarily focused on reducing the MCMF solver's algorithm runtime. To achieve low task placement latency, we must make two steps that fall outside the solver runtime efficient as well. First, Firmament must efficiently update the flow network's nodes, arcs, costs, and capacities before every MCMF optimization to reflect the chosen scheduling policy. Second, Firmament must quickly extract task placements out of the flow network after the optimization finishes. We improve over the prior work on flow-based scheduling in Quincy for both aspects, as explained in the following.

Flow network updates. Firmament does two breadth-first traversals of the flow network to update it for a new solver run. The first traversal updates resource statistics associated with every entity, such as the memory available on a machine, its current load, or a task's resource request. The traversal starts from the nodes adjacent to the sink (usually machine nodes), and propagates statistics along each node's incoming arcs. Upon the first traversal's completion, Firmament runs a second traversal that starts at the task nodes. This pass allows the scheduling policy to update the flow network's nodes, arcs, costs and capacities using the statistics gathered in the first traversal. Hence, only two passes over the large graph must be made to prepare the next solver run. Their overhead is negligible compared to the solver runtime.

Task placement extraction. At the end of a run, the solver returns an optimal flow through the given network and Firmament must extract the task placements implied by this flow. Since Firmament allows arbitrary aggregators in the flow network, paths from tasks to machines may be longer than in Quincy, where arcs necessarily pointed to machines or racks. Hence, we had to gen-

eralize Quincy's approach to this extraction [22, p. 275]. To extract task assignments efficiently, we devised the graph traversal algorithm shown in Listing 1. The algorithm starts from machine nodes and propagates a list of machines to which each node has sent flow via its incoming arcs. In the common case, the algorithm extracts the task placements in a single pass over the graph.

7 Evaluation

We now evaluate how well Firmament meets its goals:

1. How do Firmament's task placement quality and placement latency compare to Quincy's? (§7.2)
2. How does Firmament cope with demanding situations such as an overloaded cluster? (§7.3)
3. At what operating points does Firmament fail to achieve sub-second placement latency? (§7.4)
4. How does Firmament's placement quality compare to other cluster schedulers on a physical cluster running a mixed batch/service workload? (§7.5)

7.1 Methodology

Our experiments combine scale-up simulations with experiments on a local testbed cluster.

In **simulations**, we replay a public production workload trace from 12,500-machine Google cluster [30] against Firmament's implementation. Our simulator is similar to Borg's "Fauxmaster" [35, §3.1]: it runs Firmament's real code and scheduling logic against simulated machines, merely stubbing out RPCs and task execution. However, there are three important limitations to note. First, the Google trace contains multi-dimensional resource requests for each task. Firmament supports multi-dimensional feasibility checking (as in Borg [35, §3.2]), but in order to fairly compare to Quincy, we use slot-based assignment. Second, we do not enforce task constraints for the same reason, even though they typically help Firmament's MCMF solver. Third, the Google trace lacks information about job types and input sizes. We use Omega's priority-based job type classification [32, §2.1], and estimate batch task input sizes as a function of the known runtime using typical industry distributions [8].

In **local cluster experiments**, we use a homogeneous 40-machine cluster. Each machine has a Xeon E5-2430Lv2 CPU (12× 2.40GHz), 64 GB RAM, and uses a 1TB magnetic disk for storage. The machines are connected via 10 Gbps, full-bisection bandwidth Ethernet.

When we compare with **Quincy**, we run Firmament with Quincy's scheduling policy and restrict the solver to use only cost scaling (as Quincy's `cs2` solver does).

7.2 Scalability *vs.* Quincy

In Figure 3, we illustrated that Quincy fails to scale to clusters of thousands of machines at an acceptable placement latency. We now repeat the same experiment using Firmament on the full-scale simulated Google cluster. However, we increase the cluster slot utilization from the earlier experiment's 50% to 90% to make the setup more challenging for Firmament, and also tune the cost scaling-based MCMF solver for its best performance.[3]

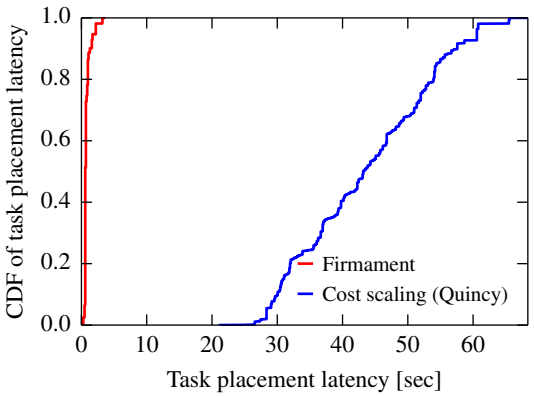


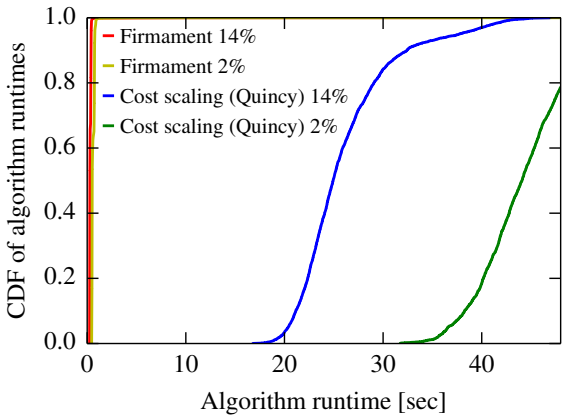
Figure 14: Firmament has a 20× lower task placement latency than Quincy on a simulated 12,500-machine cluster at 90% slot utilization, replaying the Google trace. The placement quality is identical to Quincy's.

Figure 14 shows the results as a CDF of *task placement latency*, i.e., the time between a task being submitted to the cluster manager and the time when it has been placed (§2). While Quincy takes between 25 and 60 seconds to place tasks, Firmament typically places tasks in hundreds of milliseconds and only exceeds a sub-second placement latency in the 90th percentile. Therefore, Firmament improves task placement latency by more than a 20× over Quincy, but maintains the same placement quality as it also finds an optimal flow.

Firmament's low placement latency comes because relaxation scales well even for large flow networks with the Google trace workload. This scalability allows us to afford scheduling policies with many arcs. As an illustrative example, we vary the data locality threshold in the Quincy scheduling policy. This threshold decides what fraction of a task's input data must reside on a machine or within a rack in order for the former to receive a preference arc to the latter. Quincy originally picked a threshold of a maximum of ten arcs per task. However, Figure 15a shows that even a lower threshold of 14% local data, which corresponds to at most seven preference arcs, yields algorithm runtimes of 20–40 seconds for Quincy's cost scaling. A low threshold allows the scheduler to exploit more fine-grained locality, but increases the number of arcs in the graph. Consequently, if we lower the threshold to 2% local data,[4] the cost scaling runtime in-

[3] Specifically, we found that an α-factor parameter value of 9, rather than the default of 2 used in Quincy, improves runtime by ≈30%.

[4] 2% is a somewhat extreme value used for exposition here. The benefit of such a low threshold in a real cluster would likely be limited.



(a) Low preference thresholds see subs-second runtimes in Firmament, while Quincy (with cost scaling) takes over 40s.

Pref. threshold [local data]	*Input data locality*
14%	56%
2%	71%

(b) A lower preference threshold improves data locality.

Figure 15: Firmament scales to many arcs, and thus supports a lower preference arc threshold than Quincy.

creases to well over 40 seconds. Firmament, on the other hand, still achieves sub-second algorithm even with a 2% threshold. This threshold yields an increase in data locality from 56% to 71% of total input data (Table 15b), which saves 4 TB of network traffic per simulated hour.

7.3 Coping with demanding situations

In the previous experiments, Firmament had a lower placement latency than Quincy because relaxation handles the Google workload well. As explained in §4, there are situations in which this is not the case. In those situations, Firmament picks incremental cost scaling's solution as it finishes first (§6). We now demonstrate the benefits of running two algorithms rather than just one.

In this experiment, we shrink the number of slots per cluster machine to reach 97% average utilization. Consequently, the cluster experiences transient periods of oversubscription. Figure 16 compares Firmament's automatic use of the fastest algorithm against using only one algorithm, either relaxation or cost scaling. During oversubscription, relaxation alone takes hundreds of seconds per run, while cost scaling alone completes in ≈30 seconds independent of cluster load. Firmament's incremental cost scaling finishes first in this situation, taking 10–15 seconds, which is about 2× faster than using cost scaling only (as Quincy does). Firmament also recovers earlier from the overload situation starting at 2,200s: while the relaxation-only runtime returns to sub-second level only around 3,700s, Firmament recovers at 3,200s. Relaxation on its own takes longer to recover because

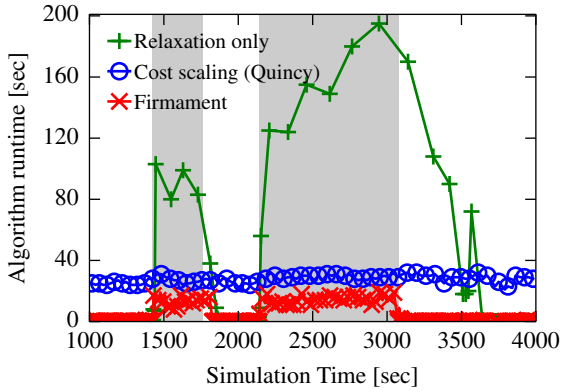


Figure 16: At times of high utilization (gray), Firmament outperforms relaxation and Quincy's cost scaling.

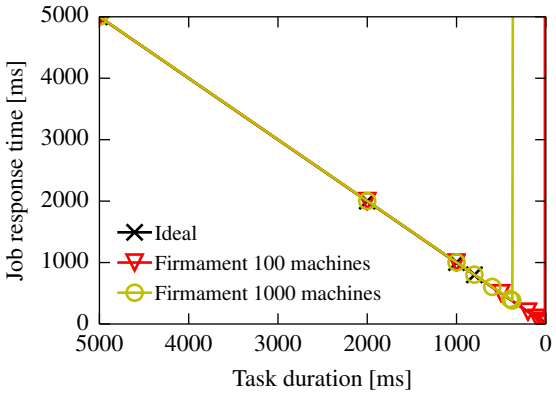


Figure 17: Firmament's breaking point is at tasks are shorter than ≈5ms at 100-machine scale, and ≈375ms at 1,000-machine scale, with 80% cluster slot utilization.

many tasks complete and free up slots during the long solver runs. These slots cannot be re-used until the next solver run completes, even though new, waiting tasks accumulate. Hence, Firmament's combination of algorithms outperforms either algorithm running alone.

7.4 Scalability to sub-second tasks

In the absence of oversubscription, we now investigate the scalability limit of Firmament's sub-second relaxation-based MCMF. To find Firmament's breaking point, we subject it to a worst-case workload consisting entirely of short tasks. This experiment is similar to Sparrow's breaking-point experiment for the centralized Spark scheduler [28, Fig. 12]. We submit jobs of 10 tasks at an interarrival time that keeps the cluster at a constant load of 80% if there is no scheduler overhead. We measure *job response time*, which is the maximum of the ten task response times for a job. In Figure 17, we plot job response time as a function of decreasing task duration. As

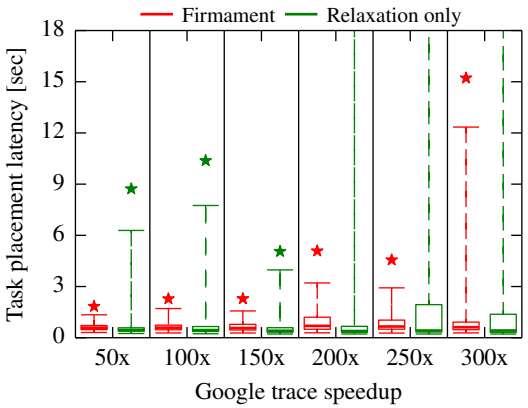


Figure 18: Firmament, unlike relaxation alone, keeps up with a 300× accelerated Google workload (1st, 25th, 50th, 75th, 99th percentiles and maximum).

we reduce task duration, we also reduce task interarrival time to keep the load constant, hence increasing the task throughput faced by the scheduler. With an ideal scheduler, job response time would be equal to task runtime as the scheduler would take no time to choose placements. Hence, the breaking point occurs when job response time deviates from the diagonal. For example, Spark's centralized task scheduler in 2013 had its breaking point on 100 machines at a 1.35 second task duration [28, §7.6].

By contrast, even though Firmament runs MCMF over the entire workload every time, Figure 17 shows that it achieves near-ideal job response time down to task durations as low as 5ms (100 machines) or 375ms (1,000 machines). This makes Firmament's response time competitive with distributed schedulers on medium-sized clusters that only run short tasks. At 10,000 machines, Firmament keeps up with task durations ≥5s. However, such large clusters usually run a mix of long-running and short tasks, rather than short tasks only [7; 10; 23; 35].

We therefore investigate Firmament's performance on a mixed workload. We speed up the Google trace by dividing all task runtimes and interarrival times by a speedup factor. This simulates a future workload of shorter batch tasks [27], while service jobs are still long-running. For example, at a 200× speedup, the median batch task takes 2.1 seconds, and the 90th and 99th percentile batch tasks take 18 and 92 seconds. We measure Firmament's placement latency across all tasks, and plot the distributions in Figure 18. Even at a speedup of 300×, Firmament keeps up and places 75% of the tasks at with sub-second latency. As before, a single MCMF algorithm does not scale: cost scaling's placement latency already exceeds 10s even without any speedup, and relaxation sees tail latencies well above 10 seconds beyond a 150× speedup, while Firmament scales further.

7.5 Placement quality on a local cluster

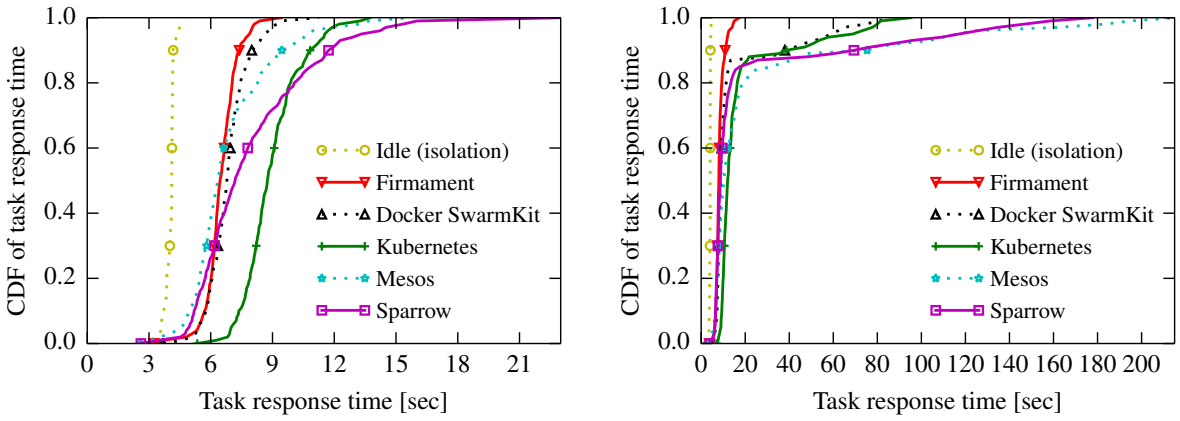
We deployed Firmament on a local 40-machine cluster to evaluate its real-world performance. We run a workload of short batch analytics tasks that take 3.5–5 seconds to complete on an otherwise idle cluster. Each task reads inputs of 4–8 GB from a cluster-wide HDFS installation in this experiment, and Firmament uses the network-aware scheduling policy. This policy reflects current network bandwidth reservations and observed actual bandwidth use in the flow network, and strives to place tasks on machines with lightly-loaded network connections. In Figure 19a, we show CDFs of task response times obtained using different cluster managers' schedulers. We measure task response time, and compare to a baseline that runs each task in isolation on an otherwise idle network. Firmament's task response time comes closest to the baseline above the 80th percentile as it successfully avoids overcommitting machines' network bandwidth. Other schedulers make random assignments (Sparrow), perform simple load-spreading (SwarmKit), or do not consider network bandwidth (Mesos, Kubernetes). Since our cluster is small, Firmament's task placement latency is inconsequential at around 5ms in this experiment.

Real-world clusters, however, run a mix of short, interactive tasks and long-running service and batch processing tasks. We therefore extend our workload with new long-running batch and service jobs to represent a similar mix. The long-running batch workloads are generated by fourteen `iperf` clients who communicate using UDP with seven `iperf` servers. Each `iperf` client generates 4 Gbps of sustained network traffic and simulates a batch job in a higher-priority network service class [20] than the short batch tasks (e.g., a TensorFlow [1] parameter server). Finally, we deploy three `nginx` web servers and seven HTTP clients as long-running service jobs. We run the cluster at about 80% network utilization, and again measure the task response time for the short batch analytics tasks. Figure 19b shows that Firmament's network-aware scheduling policy substantially improves the tail of the task response time distribution of short batch tasks. For example, Firmament's 99th percentile response time is 3.4× better than the SwarmKit and Kubernetes ones, and 6.2× better than Sparrow's. The tail matters, since the last task's response time often determines a batch job's overall response time (the "straggler" problem).

8 Related work

Many cluster schedulers exist, but Firmament is the first centralized one to offer high placement quality at sub-second placement latency on large clusters. We now briefly compare Firmament to existing schedulers.

Optimization-based schedulers. Firmament retains the same optimality as Quincy [22], but achieves much



(a) Short batch analytics tasks running on a cluster with an otherwise idle network. Overhead over "idle" due to contention.

(b) Short batch analytics tasks running on a cluster with background traffic from long-running batch and service tasks.

Figure 19: On a local 40-node cluster, Firmament improves task response time of short batch tasks in the tail using a network-aware scheduling policy, both (a) without and (b) with background traffic. Note the different x-axis scale.

lower placement latency. TetriSched [33] uses a mixed integer-linear programming (MILP) optimization and applies techniques similar to Firmament's (e.g., incremental restart from a prior solution) to reduce placement latency. Its placement quality degrades gracefully when terminated early (as required at scale), while Firmament always returns optimal solutions. Paragon [11], Quasar [12], and Bistro [16] also run expensive scoring computations (collaborative filtering, path selection), but scale the task placement by using greedy algorithms.

Centralized schedulers. Mesos [21] and Borg [35] match tasks to resources greedily; Borg's scoring uses random sampling with early termination [35, §3.4], which improves latency at the expense of placement quality. Omega [32] and Apollo [7] support multiple parallel schedulers to simplify their engineering and to improve scalability. Firmament shows that a single scheduler can attain scalability, but its MCMF optimization does not trivially admit multiple independent schedulers.

Distributed schedulers. Sparrow [28] and Tarcil [13] are distributed schedulers developed for clusters that see a high throughput of very short, sub-second tasks. In §7.4, we demonstrated that Firmament offers similarly low placement latency as Sparrow on clusters up to 1,000 machines, and beyond if only a part of the workload consists of short tasks. Mercury [23] is a hybrid scheduler that makes centralized, high-quality assignments for long tasks, and distributedly places short ones. With Firmament, we have shown that a centralized scheduler can scale even to short tasks, and that they benefit from the improved placement quality. Hawk [10] and Eagle [9] extend the hybrid approach with work-stealing and state gossiping techniques that improve placement quality; Yaq-d [29], by contrast, reorders tasks in worker-side queues to a similar end. Firmament shows that even a centralized scheduler can quickly schedule short tasks in large clusters with mixed workloads, rendering such complex compensation mechanisms largely unnecessary.

9 Conclusions

Firmament demonstrates that centralized cluster schedulers can scale to large clusters at low placement latencies. It chooses the same high-quality placements as an advanced centralized scheduler, at the speed and scale typically associated with distributed schedulers.

Firmament, our simulator, and our data sets are open-source and available from `http://firmament.io`. A Firmament scheduler plugin for Kubernetes [14] is currently under development.

Acknowledgements

We are grateful to M. Frans Kaashoek, Frank McSherry, Derek G. Murray, Rebecca Isaacs, Andrew Warfield, Robert Morris, and Pamela Delgado, as well as Jon Gjengset, Srivatsa Bhat, and the rest of MIT PDOS group for comments on drafts of this paper. We also thank Phil Gibbons, our shepherd, and the OSDI 2016 reviewers for their feedback. Their input much improved this paper.

This work was supported by a Google European Doctoral Fellowship, by NSF award CNS-1413920, and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, Nov. 2016.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition". In: *Synthesis Lectures on Computer Architecture* 8.3 (July 2013), pp. 1–154.

[4] Dimitri P. Bertsekas and Paul Tseng. "Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems". In: *Operations Research* 36.1 (Feb. 1988), pp. 93–114.

[5] Dimitri P. Bertsekas and Paul Tseng. "The Relax codes for linear minimum cost network flow problems". In: *Annals of Operations Research* 13.1 (Dec. 1988), pp. 125–190.

[6] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Hierarchical Scheduling for Diverse Datacenter Workloads". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 4:1–4:15.

[7] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 285–300.

[8] Yanpei Chen, Sara Alspaugh, and Randy Katz. "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads". In: *Proceedings of the VLDB Endowment* 5.12 (Aug. 2012), pp. 1802–1813.

[9] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. "Job-Aware Scheduling in Eagle: Divide and Stick to Your Probes". In: *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*. Santa Clara, California, USA, Oct. 2016.

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. "Hawk: Hybrid Datacenter Scheduling". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 499–510.

[11] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, Texas, USA, Mar. 2013, pp. 77–88.

[12] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, Utah, USA, Mar. 2014.

[13] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters". In: *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*. Kohala Coast, Hawaii, USA, Aug. 2015, pp. 97–110.

[14] Cloud Native Computing Foundation. *Kubernetes*. http://k8s.io; accessed 14/11/2015.

[15] Antonio Frangioni and Antonio Manca. "A Computational Study of Cost Reoptimization for Min-Cost Flow Problems". In: *INFORMS Journal on Computing* 18.1 (2006), pp. 61–70.

[16] Andrey Goder, Alexey Spiridonov, and Yin Wang. "Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 459–471.

[17] Andrew V. Goldberg. "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm". In: *Journal of Algorithms* 22.1 (1997), pp. 1–29.

[18] Andrew V. Goldberg and Michael Kharitonov. "On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem". In: *Network Flows and Matching: First DIMACS Implementation Challenge*. Ed. by D.S. Johnson and C.C. McGeoch. DIMACS series in discrete mathematics and theoretical computer science. American Mathematical Society, 1993.

[19] Andrew V. Goldberg and Robert E. Tarjan. "Finding Minimum-Cost Circulations by Successive Approximation". In: *Mathematics of Operations Research* 15.3 (Aug. 1990), pp. 430–466.

[20] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues don't matter when you can JUMP them!" In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, et al. "Mesos: A platform for fine-grained resource sharing in the data center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pp. 295–308.

[22] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: fair scheduling for distributed computing clusters". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pp. 261–276.

[23] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, et al. "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 485–497.

[24] Zoltán Király and P. Kovács. "Efficient implementations of minimum-cost flow algorithms". In: *CoRR* abs/1207.6381 (2012).

[25] Morton Klein. "A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems". In: *Management Science* 14.3 (1967), pp. 205–220.

[26] Andreas Löbel. *Solving Large-Scale Real-World Minimum-Cost Flow Problems by a Network Simplex Method*. Tech. rep. SC-96-07. Zentrum für Informationstechnik Berlin (ZIB), Feb. 1996.

[27] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, et al. "The case for tiny tasks in compute clusters". In: *Proceedings of the 14th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, New Mexico, USA, May 2013.

[28] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 69–84.

[29] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. "Efficient Queue Management for Cluster Scheduling". In: *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys)*. London, United Kingdom, Apr. 2016.

[30] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*. San Jose, California, Oct. 2012, 7:1–7:13.

[31] Malte Schwarzkopf. "Operating system support for warehouse-scale computing". PhD thesis. University of Cambridge Computer Laboratory, Oct. 2015.

[32] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 351–364.

[33] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. "TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters". In: *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys)*. London, England, United Kingdom, 2016.

[34] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 5:1–5:16.

[35] Abhishek Verma, Luis David Pedrosa, Madhukar Korupolu, David Oppenheimer, and John Wilkes. "Large scale cluster management at Google". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015.

[36] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. "CPI^2: CPU Performance Isolation for Shared Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 379–391.