Sol: Fast Distributed Computation Over Slow Networks

Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, Mosharaf Chowdhury
University of Michigan

Abstract

The popularity of big data and AI has led to many optimizations at different layers of distributed computation stacks. Despite – or perhaps, because of – its role as the narrow waist of such software stacks, the design of the execution engine, which is in charge of executing every single task of a job, has mostly remained unchanged. As a result, the execution engines available today are ones primarily designed for low latency and high bandwidth datacenter networks. When either or both of the network assumptions do not hold, CPUs are significantly underutilized.

In this paper, we take a first-principles approach toward developing an execution engine that can adapt to diverse network conditions. Sol, our *federated execution engine architecture*, flips the status quo in two respects. First, to mitigate the impact of high latency, Sol proactively assigns tasks, but does so judiciously to be resilient to uncertainties. Second, to improve the overall resource utilization, Sol decouples communication from computation internally instead of committing resources to both aspects of a task simultaneously. Our evaluations on EC2 show that, compared to Apache Spark in resource-constrained networks, Sol improves SQL and machine learning jobs by 16.4× and 4.2× on average.

1 Introduction

Execution engines form the narrow waist of modern data processing software stacks (Figure 1). Given a user-level intent and corresponding input for a job – be it running a SQL query on a commodity cluster [9], scientific simulations on an HPC environment [52], realtime stream processing [12], or training an AI/ML algorithm across many GPUs [7] – an execution engine orchestrates the execution of tasks across many distributed workers until the job runs to completion even in the presence of failures and stragglers.

Modern execution engines have primarily targeted datacenters with low latency and high bandwidth networks. The absence of noticeable network latency has popularized the *late-binding* task execution model in the control plane [10, 36, 43, 48] – pick the worker which will run a task only when the worker is ready to execute the task – which maximizes flexibility. At the same time, the impact of the network on task execution time is decreasing with increasing network bandwidth; most datacenter-scale applications today are compute- or memory-bound [7, 42]. The availability of high bandwidth has led to *tight coupling* of a task's roles to hide design complexity in the data plane, whereby the same task reads remote input and computes on it too. Late-binding before execution and tight coupling during execution work well together when the network is well-provisioned.

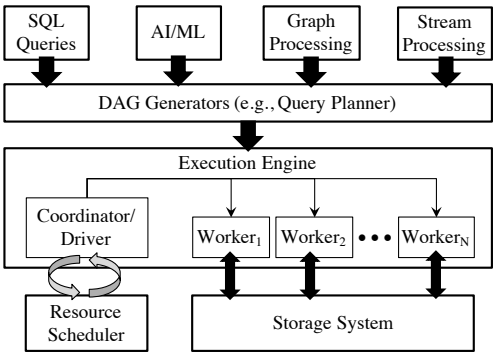


Figure 1: ***Execution engine forms the narrow waist between diverse applications and resources.***

Many emerging workloads, however, have to run on networks with high latency, low bandwidth, or both. Large organizations often perform interactive SQL and iterative machine learning between on- and off-premise storage [4, 16, 24, 27]. For example, Google uses federated model training on globally distributed data subject to privacy regulations [11, 58]; telecommunications companies perform performance analysis of radio-access networks (RAN) [30, 31]; while others troubleshoot their appliances deployed in remote client sites [39]. Although these workloads are similar to those running within a datacenter, the underlying network can be significantly constrained in bandwidth and/or latency (§2). In this paper, *we investigate the impact of low bandwidth and high latency on latency-sensitive interactive and iterative workloads*.

While recent works have proposed solutions for bandwidth-sensitive workloads, the impact of network constraints on latency-sensitive workloads has largely been overlooked. Even for bandwidth-sensitive workloads, despite many resource schedulers [28, 56], query planners [45, 50], or application-level algorithms [24, 57], the underlying execution engines of existing solutions are still primarily the ones designed for datacenters. For example, Iridium [45], Tetrium [28], and Pixida [33] rely on the execution engine of Apache Spark [54], while many others (e.g., Clarinet [50], Geode [51]) are built atop the execution engine of Apache Tez [47].

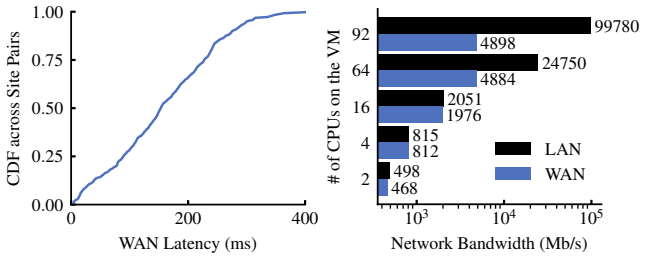
Unfortunately, under-provisioned networks can lead to large CPU underutilization in today's execution engines. First, in a high-latency network, late-binding suffers significant coordination overhead, because workers will be blocked on receiving updates from the coordinator; this leads to wasted CPU cycles and inflated completion times of latency-sensitive tasks. Indeed, late-binding of tasks to workers over the WAN can slow down the job by 8.5×–30× than running it within the local-area network (LAN). Moreover, for bandwidth-intensive tasks, coupling the provisioning of communication and computation resources at the beginning of a task's execution leads to head-of-line (HOL) blocking: bandwidth-sensitive jobs hog CPUs even though they bottleneck on data transfers, which leads to noticeable queuing delays for the rest.

By accounting for network conditions, we present a federated execution engine, Sol, which is API-compatible with Apache Spark [54]. Our design of Sol, which can transparently run existing jobs and WAN-aware optimizations in other layers of the stack, is based on two high-level insights to achieve better job performance and resource utilization.

First, we advocate *early-binding control plane decisions over the WAN* to save expensive round-trip coordinations, while continuing to late-bind workers to tasks within the LAN for the flexibility of decision making. By promoting early-binding in the control plane, we can pipeline different execution phases of the task. In task scheduling, we subscribe tasks for remote workers in advance, which creates a tradeoff: binding tasks to a remote site too early may lead to sub-optimal placement due to insufficient knowledge, but deferring new task assignments until prior tasks complete leaves workers waiting for work to do, thus underutilizing them. Our solution deliberately balances efficiency and flexibility in scheduling latency-bound tasks, while retaining high-quality scheduling for latency-insensitive tasks even under uncertainties.

Second, *decoupling the provisioning of resources for communication and computation within data plane task executions* is crucial to achieving high utilization. By introducing dedicated communication tasks for data reads, Sol decouples computation from communication and can dynamically scale down a task's CPU requirement to match its available bandwidth for bandwidth-intensive communications; the remaining CPUs can be redistributed to other jobs with pending computation.

Our evaluations show that Sol can automatically adapt to diverse network conditions while largely improving application-level job performance and cluster-level resource utilization. Using representative industry benchmarks on a 40-machine EC2 cluster across 10 regions, we show that Sol speeds up SQL and machine learning jobs by 4.9× and 16.4× on average in offline and online settings, respectively, compared to Apache Spark in resource-constrained networks. Even in datacenter environments, Sol outperforms Spark by 1.3× to 3.9×. Sol offers these benefits while effectively handling uncertainties and gracefully recovering from failures.



(a) Pairwise latency across 44 DCs. (b) Measured bandwidth on EC2.

Figure 2: ***While execution engines are widely deployed on cloud platforms, the underlying network conditions can be diverse in latency and bandwidth.***

2 Background and Motivation

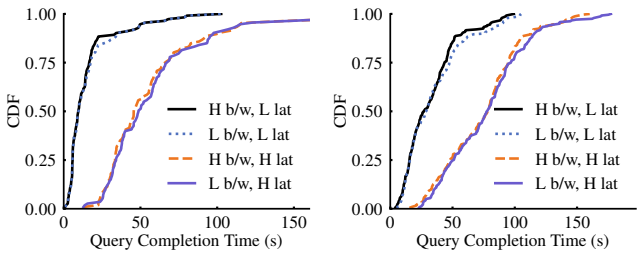
2.1 Execution Engines

The execution engine takes a graph of *tasks* – often a directed acyclic graph (DAG) – from the higher-level scheduler as its primary input. Tasks performing the same computation function on different data are often organized into *stages*, with dependencies between the stages represented by the edges of the execution DAG. Typically, a central *coordinator* in the execution engine – often referred to as the driver program of a job – interacts with the cluster resource manager to receive required resources and spawns *workers* across one or more machines to execute runnable tasks.[1] As workers complete tasks, they notify the coordinator to receive new runnable tasks to execute.

Design space. The design of an execution engine should be guided by how the environment and workload characteristics affect delays in the *control plane* (i.e., coordinations between the coordinator and workers as well as amongst the workers) and in the *data plane* (i.e., processing of data by workers). Specifically, a task's lifespan consists of four key components:

- *Coordination time* (t_{coord}) represents the time for orchestrating task executions across workers. This is affected by two factors: network latency, which can vary widely between different pairs of sites (Figure 2(a)), and the inherent computation overhead in making decisions. While the latter can be reduced by techniques like reusing schedules [37,49], the former is determined by the environment.
- *Communication time* (t_{comm}) represents the time spent in reading input and writing output of a task over the network and to the local storage.[2] For the same amount of data, time spent in communication can also vary widely based on Virtual Machine (VM) instance types and LAN-vs-WAN (Figure 2(b)).
- *Computation time* (t_{comp}) represents the time spent in running every task's computation.
- *Queuing time* (t_{queue}) represents the time spent waiting

[1] A task becomes runnable whenever its dependencies have been met.
[2] Most transfers after the input-reading stages happen over the network.



(a) Apache Spark (b) Apache Tez

Figure 3: ***TPC query completion times in different network settings using different execution engines (scale factor is set to 100).***

for resource availability before execution. Given a fixed amount of resources, tasks of one job may have to wait for tasks of other jobs to complete.

We first take into account t_{comp}, t_{coord}, and t_{comm} in characterizing the design of execution engines for a single task. By assuming $t_{comp} \gg t_{coord}, t_{comm}$ (i.e., by focusing on the execution of *compute-bound* workloads such as HPC [21], AI training [7] and in many cases within datacenters), existing execution engines have largely ignored two settings in the design space.

First, the performance of jobs can be dominated by the coordination time (i.e., $t_{coord} \gg t_{comm}, t_{comp}$), and more time is spent in the control plane than the data plane. An example of such a scenario within a datacenter would be stream processing using mini-batches, where scheduling overhead in the coordinator is the bottleneck [49]. As $t_{coord} \to O(100)$ ms over the WAN, coordination starts to play a bigger role even when scheduler throughput is not an issue. As $\frac{t_{comp}}{t_{coord}}$ and $\frac{t_{comm}}{t_{coord}}$ decrease, e.g., in interactive analytics [30, 31] and federated learning [11], coordination time starts to dominate the end-to-end completion time of each task.

Second, in *bandwidth-bound* workloads, more time is likely to be spent in communication than computation (i.e., $t_{comm} > t_{coord}, t_{comp}$). Examples of such a scenario include big data jobs in resource-constrained private clusters [36] or across globally distributed clouds [24, 45, 50, 51], and data/video analytics in a smart city [25].

In the presence of multiple jobs, inefficiency in one job's execution engine can lead to inflated t_{queue} for other jobs' tasks. For latency-sensitive jobs waiting behind bandwidth-sensitive ones, t_{queue} can quickly become non-negligible.

2.2 Inefficiencies in Constrained Network Conditions

While there is a large body of work reasoning about the performance of existing engines in high-bandwidth, low-latency datacenters [41, 42], the rest of the design space remains unexplored. We show that existing execution engines suffer significant resource underutilization and performance loss in other settings.

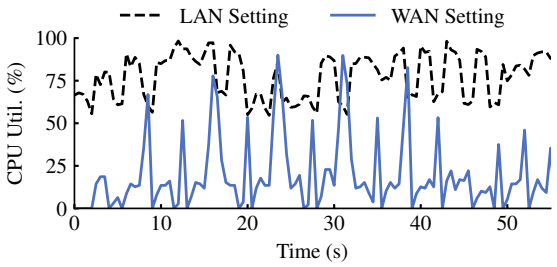


Figure 4: ***CPU utilization throughout a machine learning job.***

Performance degradation due to high latency. To quantify the impact of high latency on job performance, we analyzed the individual query completion times of 110 queries on two industrial benchmarks: TPC-DS and TPC-H. We use two popular execution engines – Apache Spark [54] and Apache Tez [47] – on a 10-site deployment; each site has 4 machines, each with 16 CPU cores and 64 GB of memory. We consider four network settings, each differing from the rest in terms of either bandwidth or latency as follows: [3]

- *Bandwidth:* Each VM has a 10 Gbps NIC in the high-bandwidth and 1 Gbps in the low-bandwidth setting.
- *Latency:* Latency across machines is <1 ms in the low-latency setting, while latencies across sites vary from $O(10)$–400 ms in the high-latency setting.

Figure 3 shows the distributions of average query completion times of Spark and Tez, where we use a dataset of scale factor 100.[4] We found that the availability of more bandwidth has little impact on query completion times; different query plans and task placement decisions in Spark and Tez did not improve the situation either. However, job completion times in the high-latency setting are significantly inflated – up to 20.6× – than those in the low-latency setting. Moreover, we observe high network latency can lead to inefficient use of CPUs for a latency-bound machine learning job (Figure 4).

The root cause behind this phenomenon is the *late-binding* of tasks to workers in the control plane. In existing execution engines, decision making in the control plane, such as task scheduling [50] and straggler mitigation [8], often requires realtime information from data plane executions, whereas data processing is initiated by control plane decisions. With high coordination latency, this leads to wasted CPU cycles as each blocks on acquiring updates from the other.

CPU underutilization due to low bandwidth. To understand the impact of low bandwidth on resource efficiency, we analyzed bandwidth-sensitive workloads using a scale factor of 1000 on the same experimental settings as above.

Figure 5 reports both the CPU and network utilizations throughout the execution of a representative query (query-

[3] We use the latency profile of 10 sites on EC2 and set a large TCP window size to reach the network capacity [34]. For the high-bandwidth setting and the low-bandwidth one, we refer to the available LAN bandwidth on *m4.10xlarge* and *m4.2xlarge* instances, respectively.

[4] A scale factor of X means a X GB dataset.

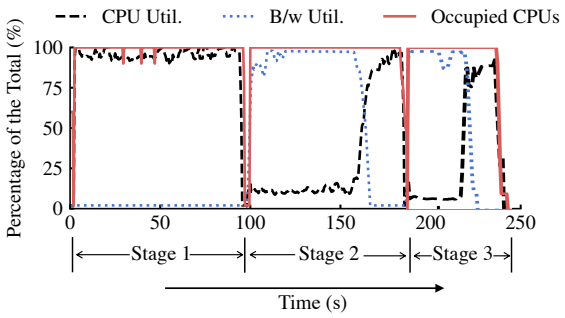


Figure 5: ***Resource utilization over a bandwidth-bound query's lifespan (scale factor is set to 1000).***

25 from the TPC-DS benchmark), which involves two large shuffles (in stage 2 and stage 3) over the network. During task executions, large data reads over the network are communication-intensive, while computations on the fetched data are CPU-intensive. We observe that, when tasks are bandwidth-constrained, their overall CPU utilization plummets even though they continue to take up all the available CPUs. This is due to the coupling of communication with computation in tasks. In other words, the number of CPUs involved in communication is independent of the available bandwidth. The end result is head-of-line (HOL) blocking of both latency- and bandwidth-sensitive jobs (not shown) by bandwidth-bound underutilized CPUs of large jobs.

Shortcomings of existing works. Existing works on WAN-aware query planning and task placement [28, 45, 50, 51] cannot address the aforementioned issues because they focus on managing and/or minimizing bandwidth usage during task execution, not on the impact of latency before execution starts or CPU usage during task execution.

3 Sol: A Federated Execution Engine

To address the aforementioned limitations, we present Sol, a *federated execution engine* which is aware of the underlying network's characteristics (Figure 6). It is primarily designed to facilitate efficient execution of emerging distributed workloads across a set of machines which span multiple sites (thus, have high latency between them) and/or are interconnected by a low bandwidth network. Sol assumes that machines within the same site are connected over a low-latency network. As such, it can perform comparably to existing execution engines when deployed within a datacenter.

Design goals. In designing Sol, we target a solution with the following properties:

- *High-latency coordinations should be pipelined.* Coordinations between control and data planes should not stall task executions. As such, Sol should avoid synchronous coordination (e.g., workers blocking to receive tasks) to reduce overall t_{coord} for latency-bound tasks. This leads to early-binding of tasks over high-latency networks.
- *Underutilized resources should be released.* Sol should release unused resources to the scheduler, which can be repurposed to optimize t_{queue} for pending tasks. This calls for decoupling communication from computation in the execution of bandwidth-intensive tasks without inflating their t_{comm} and t_{comp}.
- *Sol should adapt to diverse environments automatically.* The network conditions for distributed computation can vary at different points of the design space. Sol should, therefore, adapt to different deployment scenarios with built-in runtime controls to avoid reinventing the design.

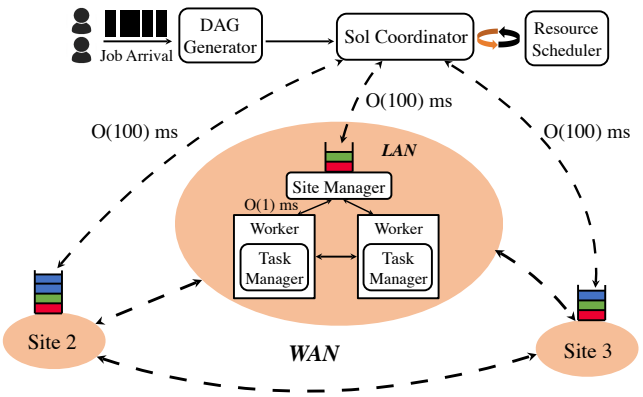


Figure 6: ***Sol components and their interactions. Low-latency sites synchronously coordinate within themselves and asynchronously coordinates across high-latency links.***

System components. At its core, Sol has three primary components:

- *Central Coordinator:* Sol consists of a logically centralized coordinator that orchestrates the input job's execution across many remote compute sites. It can be located at any of the sites; each application has its own coordinator or driver program. Similar to existing coordinators, it interacts with a resource manager for resource allocations.
- *Site Manager:* Site managers in Sol coordinate local workers within the same site. Each site manager has a shared queue, where it enqueues tasks assigned by the central coordinator to the workers in this site. This allows for late-binding of tasks to workers within the site and ensures high resource utilization, wherein decision making can inherit existing designs for intra-datacenter systems. The site manager also detects and tackles failures and stragglers that are contained within the site.
- *Task Manager:* At a high level, the task manager is the same as today: it resides at individual workers and manages tasks. However, it manages compute and communication resources independently.

Figure 7 shows a high-level overview explaining the interaction among these components throughout our design in the control plane (§4) and the data plane (§5).

4 Sol Control Plane

Modern execution engines primarily target datacenters with low latency networks [7, 12, 47, 54], wherein late-binding of

```
▷ Operations in Central Coordinator
for Site s in all sites do
  while currentTaskNum(s) < targetQueLen(s) do ▷ §4.3
    if Exist available tasks t for scheduling to s then
      Push t to Site Manager in s
    else
      Breakdown task dependency judiciously ▷ §4.4

▷ Operations in Site Manager
if Receive task assignment then
  Queue up task
else if Receive task completion then
  Notify coordinator and schedule next task t
  if Task t requires large remote read then
    Issue fetch request to the scheduled worker ▷ §5.1
  else
    Launch task t
else if Input is ready for computation task t then
  Activate and launch task t ▷ §5.3

▷ Operations in Task Manager
if Receive task assignment t then
  Execute task t
else if Detect task completion then
  Notify Site Manager for new task assignment
else if Receive data fetch request then
  Initiate communication task ▷ §5.2
```

Figure 7: ***The interaction between the central coordinator, site manager and task manager.***

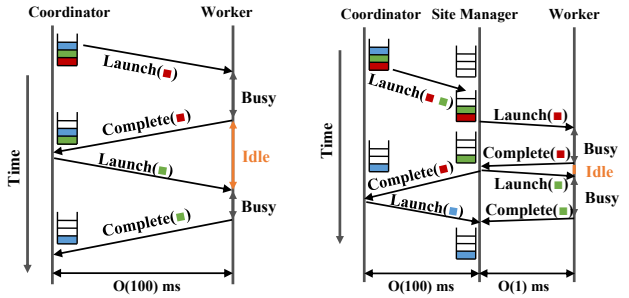
tasks to workers maximizes flexibility. For example, the coordinator assigns new tasks to a worker after it is notified of new resource availability (e.g., due to task completion) from that worker. Moreover, a variety of on-demand communication primitives, such as variable broadcasts and data shuffles, are also initiated lazily by the coordinator and workers. In the presence of high latency, however, late-binding results in expensive coordination overhead (§2.2).

In this section, we describe how Sol pushes tasks to sites to hide the expensive coordination latency (§4.1), the potential benefits of push-based execution (§4.2) as well as how we address the challenges in making it practical; i.e., how to determine the right number of tasks to push to each site (§4.3), how to handle dependencies between tasks (§4.4), and how to perform well under failures and uncertainties (§4.5).

4.1 Early-Binding to Avoid High-Latency Coordination

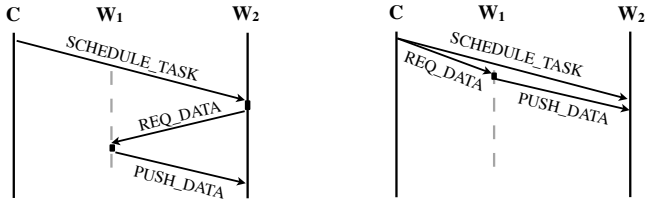
Our core idea to hide coordination latency (t_{coord}) over high-latency links is early-binding tasks to sites. Specifically, Sol optimizes t_{coord} between the central coordinator and remote workers (i) by *pushing* and queuing up tasks in each site; and (ii) by *pipelining* task scheduling and execution across tasks.

Figure 8 compares the control flow in traditional designs



(a) Traditional Design (b) Sol's Design

Figure 8: ***Task execution control flows in traditional designs vs. Sol. In Sol, tasks (denoted by colored rectangles) are queued at a site manager co-located with workers.***



(a) Pull-based Model (b) Push-based Model

Figure 9: ***Sol adopts the push-based model to pipeline task scheduling and data fetch.***

with that in Sol. In case of late-binding, workers have to wait for new task assignments from the remote coordinator. In contrast, the site manager in Sol directly dispatches a task already queued up in its local queue and asynchronously notifies the coordinator of task completions as well as the resource status of the site. The coordinator makes new task placement decisions and queues up tasks in site managers asynchronously, while workers in that site are occupied with task executions.

Furthermore, this execution model enables us to pipeline t_{coord} and t_{comm} for each individual task's execution. When the coordinator assigns a task to a site, it notifies the corresponding upstream tasks (i.e., tasks in the previous stage) of this assignment. As such, when upstream tasks complete, they can proactively push their latency-bound output partitions directly to the site where their downstream tasks will execute, even though the control messages containing task placements may still be on-the-fly. As shown in Figure 9, pull-based data fetches experience three sequential phases of communication; in contrast, the scheduling of downstream tasks and their remote data reads are pipelined in the push-based model, improving their completion times.

4.2 Why Does Early-Binding Help?

Assume that the coordinator continuously allocates tasks to a remote site's k CPUs. For each task completion, the pull-based model takes one RTT for the coordinator to receive the task completion notification and then send the next task assignment; during this period, the task is queued up in the coordinator for scheduling. Hence, on average, $\frac{i-1}{k}$ RTTs are

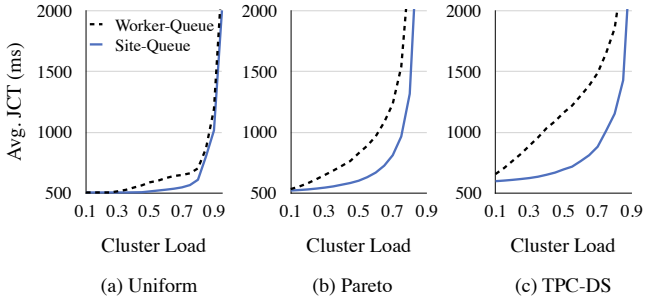


Figure 10: ***Job performance with Site- and Worker-Queue approaches. Variance in task durations increases from (a) to (c).***

wasted before the i^{th} task runs. The push-based model can save up to $\frac{i-1}{k}$ RTTs for the i^{th} task by pipelining inter-task assignments and executions. Our analysis over 44 datacenters using the measured inter-site latencies shows that, compared to late-binding, the push-based model can achieve an average improvement of 153 ms for the data fetch of every downstream task (more details in Appendix A). Such gaps become magnified at scale with a large number of small tasks, e.g., a large fan-out, latency-sensitive job.

One may consider pulling multiple tasks for individual workers at a time, but the push model provides more flexibility. Pushing from the coordinator can react to online scheduling better. When tasks arrive in an online manner, multiple tasks may not be available for pulling at a time. e.g., when a new task keeps arriving right after serving a pull request, pulling multiple tasks degenerates into pulling one by one.

Moreover, by late-binding task assignments within the site, our site-manager approach enables more flexibility than pushing tasks to individual workers (i.e., maintaining one queue per worker). To evaluate this, we ran three workloads across 10 EC2 sites, where all workloads have the same average task duration from TPC-DS benchmarks but differ in their distributions of task durations. Figure 10 shows that the site-manager approach achieves superior job performance owing to better work balance, particularly when task durations are skewed.

4.3 How to Push the Right Number of Tasks?

Determining the number of queued-up tasks for site managers is crucial for balancing worker utilization versus job completion times. On the one hand, queuing up too few tasks leads to underutilization, inflating t_{queue} due to lower system throughput. On the other hand, queuing up too many leads to sub-optimal work assignments because of insufficient knowledge when early-binding, which inflates job completion times as well (see Appendix B for more details).

Our target: Intuitively, as long as a worker is not waiting to receive work, queuing more tasks does not provide additional benefit for improving utilization. To fully utilize the resource, we expect the total execution time of the queued-up tasks will occupy the CPU before the next task assignment arrives, which is the key to strike the balance between utilization and job performance.

Our solution: When every task's duration is known, the number of queued-up tasks can adapt to the instantaneous load such that the total required execution time of the queued-up tasks keeps all the workers in a site busy, but not pushing any more to retain maximum flexibility for scheduling across sites. However, individual task durations are often highly skewed in practice [8], while the overall distribution of task durations is often stable over a short period [20, 44].

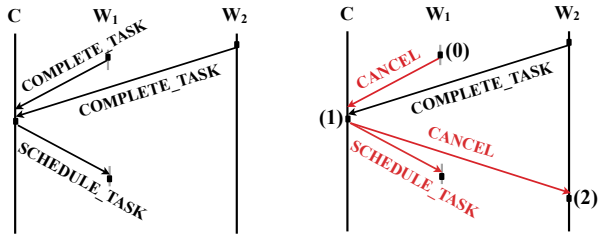
Even without presuming task-specific characteristics or distributions, we can still approximate the ideal queue length at every site dynamically for a given resource utilization target. We model the total available cycles in each scheduling round as our target, and the duration of each queued-up task is a random variable. This can be mapped into a packing problem, where we have to figure out how many random variables to sum up to achieve the targeted sum.

When the individual task duration is not available, we extend Hoeffding's inequality, and inject the utilization target into our model to determine the desired queue length (Appendix C for a formal result and performance analysis). Hoeffding's inequality is known to characterize how the sum of random variables deviates from its expected value with the minimum, the average, and the maximum of variables [23]. We extend it but filter out the outliers in tasks, wherein we rely on three statistics – 5th percentile, average, and 95th percentile (which are often stable) – of the task duration by monitoring the tasks over a period. As the execution proceeds, the coordinator in Sol inquires the model to generate the target queue size, whereby it dynamically pushes tasks to each site to satisfy the specified utilization. Note that when the network latency becomes negligible, our model outputs zero queue length as one would expect.

4.4 How to Push Tasks with Dependencies?

In the presence of task dependencies, where tasks may depend on those in their parent stage(s), pushing tasks is challenging, since it creates a tradeoff between the efficiency and quality of pipelining. For latency-sensitive tasks, we may want to push downstream tasks to save round-trip coordinations even before the upstream output is available. However, for bandwidth-intensive tasks, pushing their downstream tasks will not bring many benefits; this may even miss optimal task placements due to insufficient knowledge about the outputs from all upstream tasks [45, 50]. Sol, therefore, has to reconcile between latency- and bandwidth-sensitive tasks at runtime without presuming task properties.

To achieve desired pipelining efficiency for latency-bound tasks, Sol speculates the best placements for downstream tasks. Our straw-man heuristic is first pushing the downstream task to the site with the least work, with an aim to minimize the queueing time on the site. Moreover, Sol can refine its speculation by learning from historical trends (e.g., recurring jobs) or the iterative nature of many jobs. For example, in stream processing and machine learning, the output partitions



(a) Baseline w/o pipelining (b) Sol recovers from mistake

Figure 12: ***(b) The recovery process is shown in red. (0) W_1 detects large output, and sends CANCEL message to the coordinator C for task rescheduling. (1) Upon receiving the update, C waits until it gathers required information, then reschedules task, and (3) cancels task in W_2.***

computed for every batch are largely similar [55], so are their task placements [49]. As such, Sol can reuse the placement decisions in the past run.

However, even when a bandwidth-intensive task is pushed to a suboptimal site, Sol can gracefully retain the scheduling quality via worker-initiated re-scheduling. Figure 12 shows the control flow of the recovery process in Sol and the baseline. In Figure 12(b), we push upstream tasks to workers W_1 and W_2, and downstream tasks are pushed only to W_2. As the upstream task in W_1 proceeds, the task manager detects large output is being generated, which indicates we are in the regime of bandwidth-intensive tasks. (0) Then W_1 notifies the coordinator C of task completion and a CANCEL message to initiate rescheduling for the downstream task. (1) Upon receiving the CANCEL message, the coordinator will wait until it collects output metadata from W_1 and W_2. The coordinator then reschedules the downstream task, and (2) notifies W_2 to cancel the pending downstream task scheduled previously. Note that the computation of a downstream task will not be activated unless it has gathered the upstream output.

As such, even when tasks are misclassified, Sol performs no worse than the baseline (Figure 12(a)). First, the recovery process does not introduce more round-trip coordinations due to rescheduling, so it does not waste time. Moreover, even in the worst case, where all upstream tasks have preemptively pushed data to the downstream task by mistake, the total amount of data transfers is capped by the input size of the downstream. However, note that the output is pushed only if it is latency-sensitive, so the amount of wasted bandwidth is also negligible as the amount of data transfers is latency-bound.

4.5 How to Handle Failures and Uncertainties?

Fault tolerance and straggler mitigation are enforced by the local site manager and the global coordinator. Site managers in Sol try to restart a failed task on other local workers; failures of long tasks or persistent failures of short tasks are handled via coordination with the remote coordinator. Similarly, site managers track the progress of running tasks and selectively duplicate small tasks when their execution lags behind. Sol

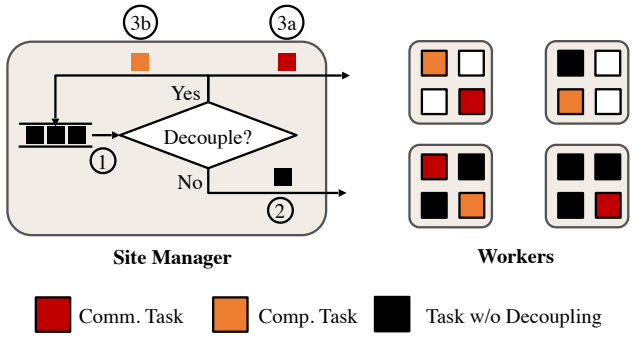


Figure 13: ***High-level overview of data plane decoupling.***

can gracefully tolerate site manager failures by redirecting workers' control messages to the central coordinator, while a secondary site manager takes over (§7.5).

Moreover, Sol can guarantee a bounded performance loss due to early-binding even under uncertainties. To ensure a task at the site manager will not experience arbitrarily large queueing delay, the site manager can withdraw the task assignment when the task queueing delay on this site exceeds Δ. As such, the total performance loss due to early-binding is (Δ + RTT), since it takes one RTT to push and reclaim the task.

5 Decoupling in the Data Plane

In existing execution engines, the amount of CPU allocated to a task when it is reading data is the same as when it later processes the data. This tight coupling between resources leads to resource underutilization (§2.2). In this section, we introduce how to improve t_{queue} by mitigating HOL blocking.

5.1 How to Decouple the Provisioning of Resource?

To remove the coupling in resource provisioning, Sol introduces dedicated communication tasks,[5] which fetch task input from remote worker(s) and deserialize the fetched data, and computation tasks, which perform the computation on data. The primary goal of decoupling is to scale down the CPU requirements when multiple tasks have to fetch data over low-bandwidth links.

Communication and computation tasks are internally managed by Sol without user intervention. As shown in Figure 13, ① when a task is scheduled for execution, the site manager checks its required input for execution and reserves the provisioned resource on the scheduled worker. ② Bandwidth-insensitive tasks will be dispatched directly to the worker to execute. ③a However, for tasks that need large volumes of remote data, the site manager will notify the task manager on the scheduled worker to set up communication tasks for data preparation. ③b At the same time, the corresponding computation tasks be marked as inactive and do not start their execution right away. Once input data is ready for computation, the site manager will activate corresponding computation tasks to perform computation on the fetched data.

[5] Each communication task takes one CPU core by default in our design.

Although decoupling the provisioning of computation and communication resource will not speed up individual tasks, it can greatly improve overall resource utilization. When the input for a task's computation is being fetched by the communication task, by oversubscribing multiple computation tasks' communication to fewer communication tasks, Sol can release unused CPUs and repurpose them for other tasks. In practice, even the decoupled job can benefit from its own decoupling; e.g., when tasks in different stages can run in parallel, which is often true for jobs with complicated DAGs, computation tasks can take up the released CPUs from other stages in decoupling.

5.2 How Many Communication Tasks to Create?

Although decoupling is beneficial, we must avoid hurting the performance of decoupled jobs while freeing up CPUs. A key challenge in doing so is to create the right number of communication tasks to fully utilize the available bandwidth. Creating too many communication tasks will hog CPUs, while creating too few will slow down the decoupled job.

We use a simple model to characterize the number of required communication tasks. There are two major operations that communication tasks account for: (i) fetch data with CPU cost $C_{I/O}$ every time unit; (ii) deserialize the fetched data simultaneously with CPU cost C_{deser} in unit time. When the decoupling proceeds with I/O bandwidth B, the total requirement of communication tasks N can be determined based on the available bandwidth ($N = B \times (C_{I/O} + C_{deser})$).

Referring to the network throughput control, we use an adaptive tuning algorithm. When a new task is scheduled for decoupling, the task manager first tries to hold the provisioned CPUs to avoid resource shortage in creating communication tasks. However, the task manager will opportunistically cancel the launched communication task after its current fetch request completes, and reclaim its CPUs if launching more communication tasks does not improve bandwidth utilization any more.[6] During data transfers, the task manager monitors the available bandwidth using an exponentially weighted moving average (EWMA).[7] As such, the task manager can determine the number of communication tasks required currently: $N_{current} = \left\lceil \frac{B_{current}}{B_{old}} \times N_{old} \right\rceil$. Therefore, it will launch more communication tasks when more bandwidth is available and the opposite when bandwidth decreases. Note that the number of communication tasks is limited by the total provisioned CPUs for that job to avoid performance interference.

5.3 How to Recover CPUs for Computation?

Sol must also ensure that the end-to-end completion time on computation experiences negligible inflation. This is because when the fetched data is ready for computation, the decoupled job may starve if continuously arriving computation tasks take up its released computation resources.

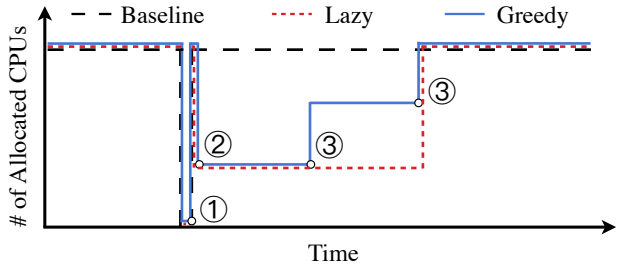


Figure 14: ***Three strategies to manage decoupled jobs. We adopt the Greedy strategy: ① when the downstream tasks start, they hold the reserved CPUs. ② But the task manager will reclaim the unused CPUs, and ③ activate the computation task once its input data is ready. The first trough marks the stage boundary.***

We refer to not decoupling as the *baseline* strategy, while waiting for the entire communication stage to finish as the *lazy* strategy. The former wastes resources, while the latter can hurt the decoupled job. Figure 14 depicts both.

Instead, Sol uses a *greedy* strategy, whereby as soon as some input data becomes ready (from upstream tasks), the site manager will prioritize the computation task corresponding to that data over other jobs and schedule it. As such, we can gradually increase its CPU allocation instead of trying to acquire all at once or holding onto all of them throughout.

5.4 Who Gets the Freed up CPUs?

Freed up CPUs from the decoupled jobs introduce an additional degree of freedom in scheduling. Resource schedulers can assign them in a FIFO or fair manner. As the duration of communication tasks can be estimated by the remaining data fetches and the available bandwidth, the scheduler can plan for the extra resources into the future, e.g., similar to [19].

6 Implementation

While our design criteria are not married to specific execution engines, we have implemented Sol in a way that keeps it API compatible with Apache Spark [2] in order to preserve existing contributions in the big data stack.

Control and Data Plane To implement our federated architecture, we add site manager modules to Spark, wherein each site manager keeps a state store for necessary metadata in task executions, and the metadata is shared across tasks to avoid redundant requests to the remote coordinator. The central coordinator coordinates with the site manager by heartbeat as well as the piggyback information in task updates. During executions, the coordinator monitors the network latency using EWMA in a one second period. This ensures that we are stable despite transient latency spikes. When the coordinator schedules a task to the site, it assigns a dummy worker for the pipelining of dependent tasks (e.g., latency-bound output will be pushed to the dummy worker). Similar to delay scheduling, we set the queueing delay bound Δ to 3 seconds [56]. Upon receiving the completion of upstream tasks, the site manager

[6] This introduces little overhead, since the data fetch is in a streaming manner, wherein the individual block is small.

[7] $B_{current} = \alpha\, B_{measured} + (1-\alpha)\, B_{old}$, where α is the smoothing factor ($\alpha = 0.2$ by default) and B denotes the available bandwidth over a period.

can schedule the downstream task more intelligently with late-binding. Meanwhile, the output information from upstream tasks is backed up in the state store until their completions.

Support for Extensions Our modifications are to the core of Apache Spark, so users can enjoy existing Spark-based frameworks on Sol without migrations of their codebase. Moreover, for recent efforts on WAN-aware optimizations, Sol can support those more educated resource schedulers or location-conscious job schedulers by replacing the default, but further performance analysis of higher-layer optimizations is out of the scope of this paper. To the best of our knowledge, Sol is the first execution engine that can optimize the execution layer across the design space.

7 Evaluation

In this section, we empirically evaluate Sol through a series of experiments using micro and industrial benchmarks. Our key results are as follows:

- Sol improves performance of individual SQL and machine learning jobs by 4.9×–11.5× w.r.t. Spark and Tez execution engines in WAN settings. It also improves streaming throughput by 1.35×–3.68× w.r.t. Drizzle (§7.2).
- In online experiments, Sol improves the average job performance by 16.4× while achieving 1.8× higher utilization (§7.3).
- Even in high bandwidth-low latency (LAN) setting, Sol improves the average job performance by 1.3× w.r.t. Spark; its improvement in low bandwidth-low latency setting is 3.9× (§7.4).
- Sol can recover from failures faster than its counterparts, while effectively handling uncertainties (§7.5).

7.1 Methodology

Deployment Setup We first deploy Sol in EC2 to evaluate individual job performance using instances distributed over 10 regions.[8] Our cluster allocates 4 *m4.4xlarge* instances in each region. Each has 16 vCPUs and 64GB of memory. To investigate Sol performance on multiple jobs in diverse network settings, we set up a 40-node cluster following our EC2 setting, and use *Linux Traffic Control* to perform network traffic shaping to match our collected profile from 10 EC2 regions.

Workloads We use three types of workloads in evaluations:

1. *SQL*: we evaluate 110 industry queries in TPC-DS/TPC-H benchmarks [5, 6]. Performance on them is a good demonstration of how good Sol would perform in real-world applications handling jobs with complicated DAGs.
2. *Machine learning*: we train three popular federated learning applications: linear regression, logistic regression, and k-means, from Intel's industry benchmark [26]. Each training data consists of 10M samples, and the training time of each iteration is dominated by computation.
3. *Stream processing*: we evaluate the maximum throughput that an execution design can sustain for *WordCount* and *TopKCount* while keeping the end-to-end latency by a given target. We define the end-to-end latency as the time from when records are sent to the system to when results incorporating them appear.

Baselines We compare Sol to the following baselines:

1. *Apache Spark* [54] and *Apache Tez* [47]: the mainstream execution engines for generic workloads in datacenter and wide-area environments.
2. *Drizzle* [49]: a recent engine tailored for streaming applications, optimizing the scheduling overhead.

Metrics Our primary metrics to quantify performance are the overarching user-centric and operator-centric objectives, including *job completion time (JCT)* and *resource utilization*.

7.2 Performance Across Diverse Workloads in EC2

In this section, we evaluate Sol's performance on individual jobs in EC2, with query processing, machine learning, and streaming benchmarks.

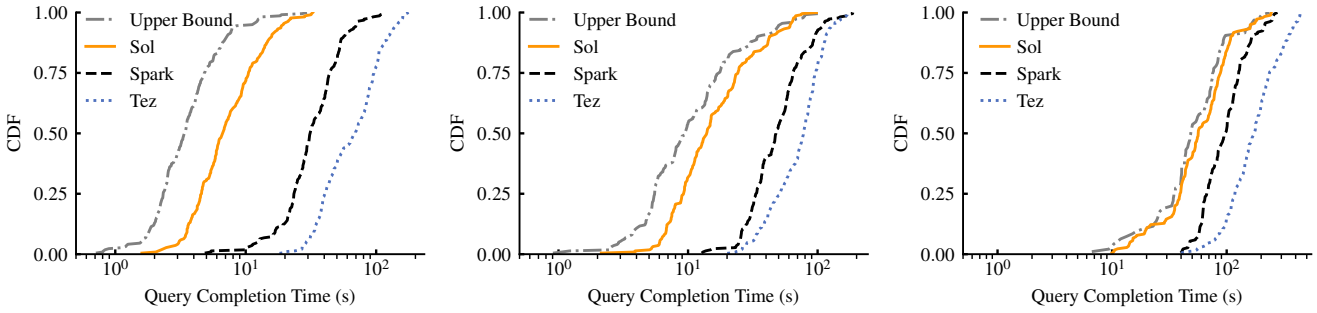
Sol outperforms existing engines Figure 15 shows the distribution of query completion times of 110 TPC queries individually on (10, 100, 1000) scale factor datasets. As expected, Sol and Spark outperform Tez by leveraging their in-memory executions. Meanwhile, Sol speeds up individual queries by 4.9× (11.5×) on average and 8.6× (23.3×) at the 95th percentile over Spark (Tez) for the dataset with scale factor 10. While these queries become more bandwidth- and computation- intensive as we scale up the dataset, Sol can still offer a noticeable improvement of 3.4× and 1.97× on average compared to Spark on datasets with scale factors 100 and 1000, respectively. More importantly, Sol outperforms the baselines across all queries.

Sol also benefits machine learning and stream processing. For such predictable workloads, Sol pipelines the scheduling and data communication further down their task dependencies. Figure 16 reports the average duration across 100 iterations in machine learning benchmarks, where Sol improves the performance by 2.64×-3.01× w.r.t. Spark.

Moreover, Sol outperforms Drizzle [49] in streaming workloads. Figure 17 shows that Sol achieves 1.35×–3.68× higher throughput than Drizzle. This is because Sol follows a *push-based* model in both control plane coordinations and data plane communications to pipeline round-trips for inter-site coordinations, while Drizzle optimizes the coordination overhead between the coordinator and workers. Allowing a larger target latency improves the throughout, because the fraction of computation time throughout the task lifespan increases, and thus the benefits from Sol become less relevant.

Sol is close to the upper bound performance To explore how far Sol is from the optimal, we compare Sol's perfor-

[8] California, Sydney, Oregon, Ohio, Tokyo, Mumbai, Seoul, Singapore, Sao Paulo and Frankfurt.



(a) Scale Factor = 10 (b) Scale Factor = 100 (c) Scale Factor = 1000

Figure 15: ***Performance of Sol, Spark, and Tez on TPC query processing benchmark.***

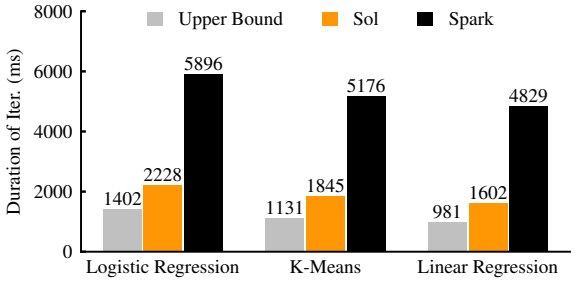


Figure 16: ***Performance on machine learning.***

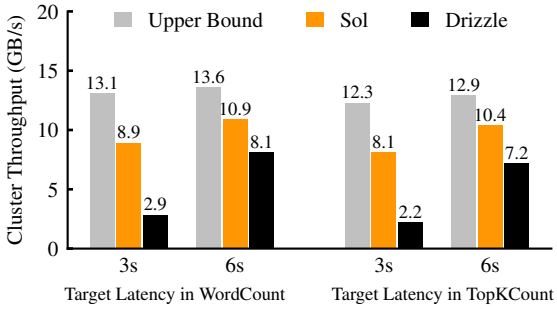
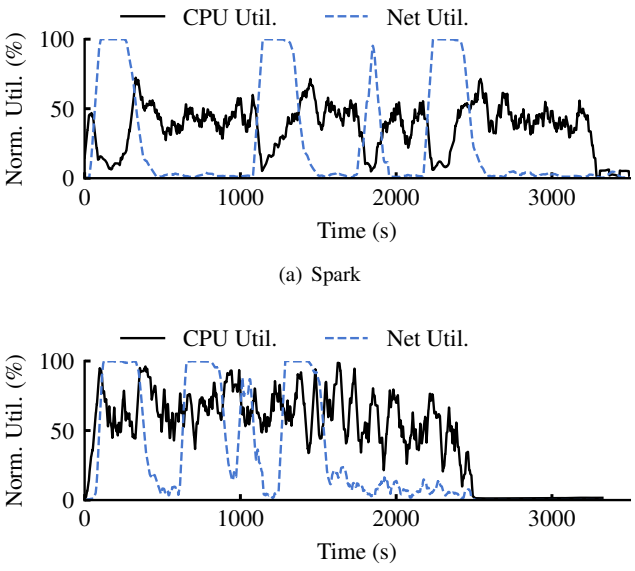


Figure 17: ***Performance on stream processing. Higher is better.***



(a) Spark

(b) Sol

Figure 18: ***Resource utilization over time.***

mance in the high latency setting against Sol in a hypothetical latency-free setting [9], which is a straightforward upper bound on its performance. While high latencies lead to an order-of-magnitude performance degradation on Spark, Sol is effectively approaching the optimal. As shown in Figure 15 and Figure 16, Sol's performance is within 3.5× away from the upper bound. As expected in Figure 15(c), this performance gap narrows down as Sol has enough work to queue-up for hiding the coordination delay.

7.3 Online Performance Breakdown

So far we have evaluated Sol in the offline setting with individual jobs. Here we move on to evaluate Sol with diverse workloads running concurrently and arriving in an online fashion with our cluster. Specifically, we evaluate Sol in an online setting, where we run 160 TPC queries – randomly drawn from the (10, 100)-scale TPC benchmarks – run as foreground, interactive jobs, and bandwidth-intensive CloudSort jobs [3] – each has 200 GB or 1 TB GB input – in the background. The TPC queries are submitted following a Poisson process with an average inter-arrival time of 10 seconds, while the CloudSort jobs are submitted every 300 seconds.

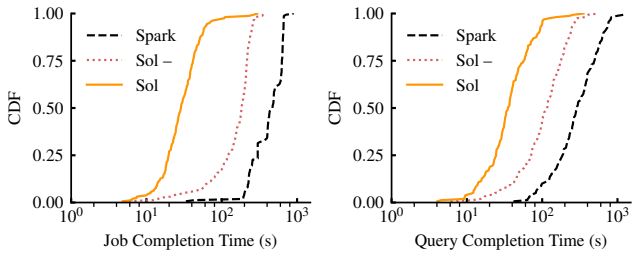
We evaluate Sol and Spark using two job schedulers:

1. *FIFO*: Jobs are scheduled in the order of their arrivals, thus easily resulting in Head-of-Line (HOL) blocking;
2. *Fair sharing*: Jobs get an equal share of resources, but the execution of early submitted jobs will be prolonged.

These two schedulers are prevalent in real deployments [1,22], especially when job arrivals and durations are unpredictable.

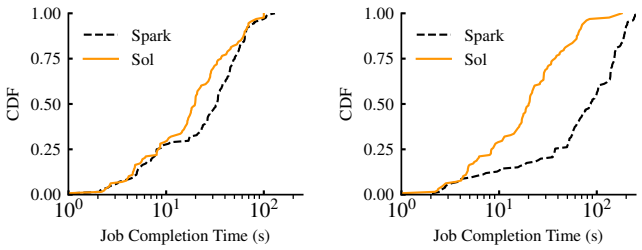
Improvement of resource utilization Figure 18 shows a timeline of normalized resource usage for both network bandwidth and total CPUs with the FIFO scheduler. A groove in CPU utilization and a peak in network utilization dictate the execution of bandwidth-intensive background jobs. Similarly, a low network utilization but high CPU utilization implicate

[9] We create a 40-node cluster in a single EC2 region.



(a) FIFO Scheduler (b) Fair Scheduler

Figure 19: ***JCT with online job arrival using different cluster schedulers. Sol- is Sol without data plane decoupling.***



(a) Low Latency and High B/w (b) Low Latency and Low B/w

Figure 20: ***Sol performance in other design space.***

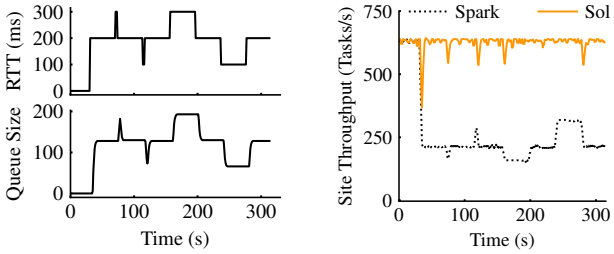


Figure 21: ***Sol performance under latency variations.***

the execution of foreground jobs. We observe Sol improves the CPU utilization by 1.8× over Spark. The source of this improvement comes from both control and data planes: (i) Sol pipelines high-latency coordinations, and thus workers are busy in running tasks all the time. (ii) Sol flexibly repurposes the idle CPU resources in the presence of bandwidth-intensive jobs, thus achieving higher utilizations by orchestrating all jobs. Note that the CPU resource is not always fully saturated in this evaluation, because the cluster is not extremely heavy-loaded given the arrival rate. Therefore, we believe Sol can provide even better performance with heavy workloads, wherein the underutilized resource can be repurposed for more jobs with decoupling. Results were similar for the fair scheduler too.

Improvement of JCTs Figure 19(a) and Figure 19(b) report the distribution of job completion times with FIFO and fair schedulers respectively. The key takeaways are the following. First, simply applying different job schedulers is far from optimal. With the FIFO scheduler, when CloudSort jobs are running, all the frontend jobs are blocked as background jobs hog all the available resources. While the fair scheduler mitigates such job starvation by sharing resource across jobs, it results in a long tail as background jobs are short of resources.

Instead, Sol achieves better job performance by improving both the intra-job and inter-job completions in the task execution level: (i) Early-binding in the control plane improves small jobs, whereby the cluster can finish more jobs in a given time. Hence, even the simple pipelining can achieve an average improvement of 2.6× with the FIFO scheduler and 2.5× with the fair scheduler. (ii) With data plane decoupling, the latency-sensitive jobs can temporarily enjoy under-utilized resource without impacting the bandwidth-intensive jobs. We observe the performance loss of bandwidth-intensive job is less than 0.5%. As the latency-sensitive jobs complete faster, bandwidth-intensive jobs can take up more resource. As such, Sol further improves the average JCTs w.r.t. Spark with both FIFO (average 16.4×) and fair schedulers (average 8.3×).

7.4 Sol's Performance Across the Design Space

We next rerun the prior online experiment to investigate Sol's performance in different network conditions with our cluster.

High bandwidth-low latency network In this evaluation, each machine has 10 Gbps bandwidth, and the latency across machines is <1 ms. Figure 20 shows the distribution of JCTs. The benefits of data plane decoupling depend on the time spent on data exchanges over the network. Although jobs are prone to finishing faster in this favorable environments, Sol can still improve over Spark by 1.3× on average by mitigating the HOL blocking with the decoupling in task executions.

Low bandwidth-low latency network In practice, users may deploy cheap VMs to perform time-insensitive jobs due to budgetary constraints. We now report the JCT distribution in such a setting, where each machine has 1 Gbps low bandwidth and negligible latency. As shown in Figure 20(b), Sol largely outperforms Spark by 3.9×. This gain is again due to the presence of HOL blocking in Spark, where bandwidth-intensive jobs hog their CPUs when tasks are reading large output partitions over the low-bandwidth network.

Note that the high latency-high bandwidth setting rarely exists. As such, Sol can match or achieve noticeable improvement over existing engines across all practical design space.

7.5 Sol's Performance Under Uncertainties

As a network-aware execution engine, Sol can tolerate different uncertainties with its federated design.

Uncertainties in network latency While Sol pushes tasks to site managers with early-binding under high network latency, its performance is robust to latency jitters. We evaluate Sol by continuously feeding our cluster with inference jobs; each scans a 30 GB dataset and the duration of each task is around 100 ms. We snapshot a single site experiencing tran-

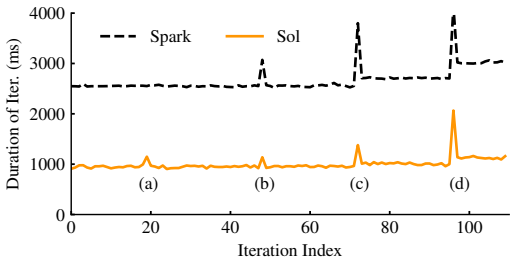


Figure 22: ***Impact of different failures on iteration duration for Sol and Spark: (a) Sol site manager failure, (b) task failure, (c) node failure , and (d) site-wide failure.***

sient or lasting latency variations. As shown in Figure 21, Sol proceeds more tasks than Spark with early-binding of tasks. Moreover, Sol can efficiently react to RTT variations by adaptively tuning its queue size.

Uncertainties in failure Figure 22 compares Sol's performance with Spark under different failures. In this evaluation, we train a long running linear regression in our 10-site deployment, and each iteration performs two stages: training on the data and the aggregation of updates. When the site manager fails (a), Sol restarts the site manager on other local machines, and reschedules the missing queued-up tasks. The recovery of site managers is pipelined with task executions, experiencing little overhead in job performance. Task failures (b) and machine failures (c) in Spark require a tight coordination with the remote coordinator, but Sol handles such failures by coordinating the site manager. Upon detecting task failures, the site manager restarts the task on other locally available machines with its metadata, while asynchronously notifying the coordinator. As such, Sol suffers little overhead by hiding the failures silently. Although the coordinator needs to take charge of rescheduling in both Sol and Spark under site-wide failures (d), tasks in Sol complete faster.

8 Discussion and Future Work

Fine-grained queue management. By capturing the range of task durations, Sol pushes the right number of tasks to site managers at runtime. However, Hoeffding's inequality can be suboptimal, especially when the variance of task durations becomes much greater than their average [23]. Further investigations on the queue management of site managers are needed. To this end, one possible approach is to build a context-aware machine learning model (e.g., reinforcement learning) to decide the optimal queue length [13].

Performance analysis of geo-aware efforts. As the first federated execution engine for diverse network conditions, Sol can serve a large body of existing efforts for geo-distributed data analytics [28, 45, 50]. Although these works do not target latency-bound tasks, for which Sol shows encouraging improvements with control plane optimizations, it would be interesting to investigate Sol's improvement for bandwidth-intensive workloads after applying techniques from existing geo-distributed frameworks.

9 Related Work

Geo-distributed storage and data analytics Numerous efforts strive to build frameworks operating on geo-distributed data. Recent examples include geo-distributed data storage [35, 53] and data analytics frameworks [4, 24]. Geode [51] aims at generating query plans that minimize data transfers over the WAN, while Clarinet [50] and Iridium [45] develop the WAN-aware query optimizer to optimize query response time subject to heterogeneous WAN bandwidth. These optimizations for data analytics lie on the scheduler layer and could transparently leverage Sol for further gains (§6).

Data Processing Engines The explosion of data volumes has fostered the world of MapReduce-based parallel computations [18]. Naiad [40] and Flink [12] express data processing as pipelined fault-tolerant data flows, while the batch processing on them performs similar to Spark [54]. The need for expressive user-defined optimizations motivates Dryad [29] and Apache Tez [47] to enable runtime optimizations on execution plans. These paradigms are designed for well-provisioned networks. Other complementary efforts focus on reasoning about system performance [41, 42], or decoupling communication from computation to further optimize data shuffles [14, 15]. Our work bears some resemblance, but our focus is on designing a network-aware execution engine.

Speeding up data-parallel frameworks Although Nimbus [37] and Drizzle [49] try to speed up execution engines, they focus on amortizing the computation overhead of scheduling for iterative jobs. Hydra [17] democratizes the resource management for jobs across multiple groups. While Yaq-c [46] discusses the tradeoff between utilization and job performance in queue management, its solution is bound to specific task durations without dependencies. Moreover, we optimize task performance inside execution engines.

10 Conclusion

As modern data processing expands due to changing workloads and deployment scenarios, existing execution engines fall short in meeting the requirements of diverse design space. In this paper, we explored the possible designs beyond those for datacenter networks and presented Sol, a federated execution engine that emphasizes an early-binding design in the control plane and decoupling in the data plane. In comparison to the state-of-the-art, Sol can match or achieve noticeable improvements in job performance and resource utilization across all practical points in the design space.

Acknowledgments

Special thanks go to the entire CloudLab team for making Sol experiments possible. We would also like to thank the anonymous reviewers, our shepherd, Ramachandran Ramjee, and SymbioticLab members for their insightful feedback. This work was supported in part by NSF grants CNS-1563095, CNS-1617773, and CNS-1900665.

References

[1] Apache Hadoop NextGen MapReduce (YARN). http://goo.gl/etTGA.

[2] Apache spark. https://spark.apache.org/.

[3] Sort Benchmark. http://sortbenchmark.org/.

[4] TensorFlow Federated. https://www.tensorflow.org/federated.

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

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

[7] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, and et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[8] G. Ananthanarayanan, A. Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[9] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[10] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *EuroSys*, 2018.

[11] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, and et al. Towards federated learning at scale: System design. In *SysML*, 2019.

[12] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[13] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM*, 2018.

[14] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[15] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):8, 2013.

[17] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, and et al. Hydra: a federated resource manager for datacenter scale analytics. In *NSDI*, 2019.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[19] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[20] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[21] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

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

[23] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *Journal of the American Statistical Association*, 1963.

[24] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. Ganger, P. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*, 2017.

[25] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[26] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshop*, 2010.

[27] Yuzhen Huang, Yingjie Shi, Zheng Zhong, and et al. Yugong: Geo-Distributed data and job placement at scale. In *VLDB*, 2019.

[28] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *EuroSys*, 2018.

[29] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[30] Anand Padmanabha Iyer, Li Erran Li, Mosharaf Chowdhury, and Ion Stoica. Mitigating the latency-accuracy trade-off in mobile data analytics systems. In *MobiCom*, 2018.

[31] Anand Padmanabha Iyer, Li Erran Li, and Ion Stoica. Celliq : Real-time cellular network analytics at scale. In *NSDI*, 2015.

[32] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 745–761, Carlsbad, CA, October 2018. USENIX Association.

[33] Konstantinos Kloudas, Margarida Mamede, Nuno Preguica, and Rodrigo Rodrigues. Pixida: Optimizing data parallel jobs in wide-area data analytics. In *VLDB*, 2015.

[34] Fan Lai, Mosharaf Chowdhury, and Harsha Madhyastha. To relay or not to relay for inter-cloud transfers? In *HotCloud*, 2018.

[35] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[36] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using QOOP. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 253–267, Carlsbad, CA, 2018. USENIX Association.

[37] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *ATC*, 2017.

[38] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[39] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *ACM SIGCOMM Computer Communication Review*, 2015.

[40] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[41] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.

[42] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[43] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[44] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *EuroSys*, 2018.

[45] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Victor Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[46] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *EuroSys*, 2016.

[47] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curinom. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, 2015.

[48] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[49] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, and Michael J. Franklin. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[50] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[51] Ashish Vulimiri, Carlo Curino, B Godfrey, J Padhye, and G Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[52] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[53] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

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

[55] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[56] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[57] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. Awstream: Adaptive wide-area streaming analytics. In *SIGCOMM*, 2018.

[58] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

A Benefits of Pipelining

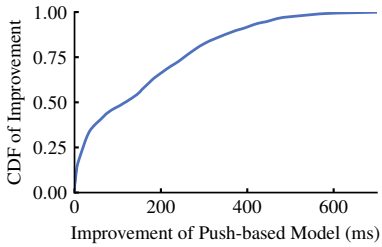


Figure 23: ***Improvement of Push-based Model in (§4.2).***

To investigate the benefit of pipelining the scheduling and data fetch of a downstream task, we assume zero queuing time and emulate the inter-site coordinations with our measured latency across 44 datacenters on AWS, Azure and Google Cloud. We define the improvement as the difference between the duration of the pull-based model and that of our proposed push-based model for every data fetch (Figure 9). Our results in Figure 23 report we can achieve an average improvement of 153 ms.

Understandably, such benefit is more promising in consideration of the task queuing time, as pushing the remote data is even pipelined with the task spin-wait for scheduling.

B Impact of Queue Length

We quantify the impact of queue size with the aforementioned three workloads (in (§4.2)). In this experiment, we analyze three distinct sites, where the network latency from Site A, B and C to the centralized coordinator is 130 ms, 236 ms and 398 ms, respectively. As shown in Figure 24, queuing up too many or too few tasks can hurt job performance.

C Determining the Queue Length

Lemma 1. *For a given utilization level* δ *and confidence interval* α *(i.e.,* $\Pr[Util. > \delta] > \alpha$*), the queue length K for S working slots satisfies:*

$$K \geq M - \frac{C}{2} + \frac{1}{2D_{avg}}\sqrt{(CD_{avg})^2 - 4MCD_{avg}} \qquad (1)$$

where D_{5th} *and* D_{95th} *denote the 5th and 95th percentile of task durations respectively, and* D_{avg} *denotes the average task duration.* $M = \frac{\delta RTT \times S}{D_{avg}}$, $C = \frac{1}{2}\left(\frac{D_{95th} - D_{5th}}{D_{avg}}\right)^2 \cdot \log\alpha$. *The first term M depicts the expectation, while the rest capture the skewness of distributions and confidence.*

This is true for any distribution of task durations. Unfortunately, we omit the proof for brevity. D_{avg}, D_{5th} and D_{95th} are often stable in a large cluster, and thus available from the historical data. α is the configurable confidence level, which is often set to 99% [32, 38], and δ is set to $\geq 100\%$ to guarantee full utilization. Note that from Eq. 1, when task durations follow the uniform distribution, our model ends up with the expectation M. Similarly, when the RTT becomes negligible, this outputs zero queue length.

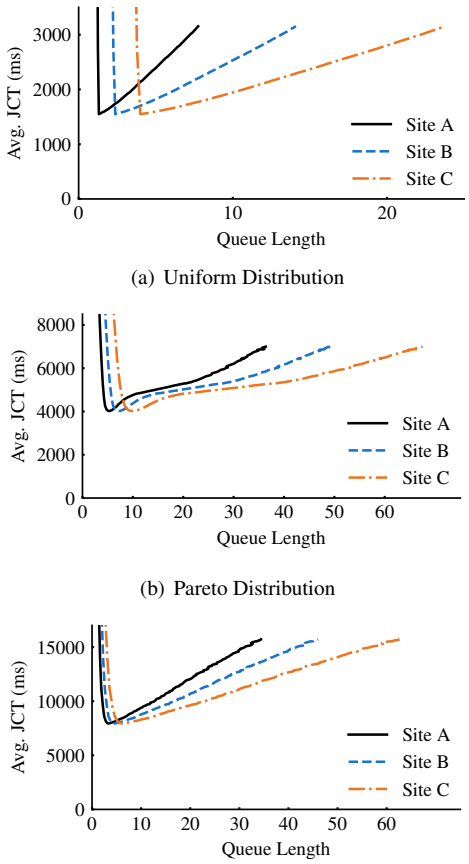


Figure 24: ***Impact of Queue Size on Job Completion Time (JCT).***

Our evaluations show this model can provide encouraging performance, wherein we reran the prior experiments with the workloads mentioned above (§4.3). We provide the results for workloads with Pareto and TPC-DS distributions by injecting different utilizations in theory, since results for the Uniform distribution are concentrated on a single point (i.e., the expectation M). As shown in Figure 25, the queue length with 100% utilization target locates in the sweet spot of JCTs.

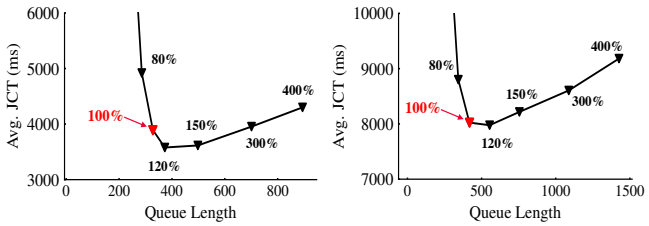


Figure 25: ***JCT performance with different utilization targets.***

Note that when more task information is available, one can refine this range better; e.g., the bound of Eq. (1) can be improved with Chernoff's inequality when the distribution of task durations is provided.