PushBox: Making Use of Every Bit of Time to Accelerate Completion of Data-Parallel Jobs

Chen Tian, Yi Wang, Bingchuan Tian, Yang Zhao, Yuhang Zhou, Chenxu Wang, Haoran Guan, Wanchun Dou, and Guihai Chen

Abstract—To minimize a job's completion time, we need to minimize the completion time of its final stage's last task. Scheduling of machine slots and networks largely dominates the variable part of each task's duration. Finding an optimal schedule is NP-hard even for offline and simplified scenarios. Previous work does lead to improved performance with various strategies. State-of-the-art task placement and network scheduling efforts are largely disjunctive. Without joint optimization, they are sub-optimal and myopic in many scenarios. Task placement usually treats the network as a black box. Thus, we use prioritized bandwidth allocation among tasks making the network both *predictable* and *efficient* to achieve joint scheduling. With this feature, joint scheduling can be transformed into a special *bin-packing problem*. Over this minimal yet power-enough abstraction, we propose PushBox to schedule data-parallel jobs in multi-tenant clusters. When designing the joint scheduling algorithm, we not only embrace the wisdom of prior art but also respect administrators' fairness intent, which is so far largely ignored. We implement PushBox on Hadoop 3. PushBox performs persistently well on both a small testbed and a trace-driven simulator.

Index Terms—Task scheduling, distributed system, datacenter

1 Introduction

To provide a small response time for interactive data-analytic queries [1], cluster operators rely on in-memory data-parallel frameworks (e.g., in-memory MapReduce [2], [3], Spark [4]). As shown in Fig. 1a, a data-parallel job in such a framework can be defined as a directed graph of processing *stages*. An edge defines the dependency between a preceding stage and the following stage. Each stage is comprised of many independent execution *tasks*. Cluster machines divide their computation capacity into several *slots* (e.g., one CPU core, one Docker container, *etc.*) to execute these tasks. A task execution example is demonstrated in Fig. 1b, where stage A/B/C/D has 3/2/2/2 tasks respectively. A solid line represents an individual task running in a slot, and an arrow represents data dependency between two tasks. A stage's completion is marked by its last task's completion. To minimize a job's completion time, we need to minimize the completion time of its final stage's last task.

To facilitate analysis, we present a conceptual model of a task's lifetime in Fig. 1c. There are 4 phases. All tasks of a job (virtually) start their *admitting* phases at the job submission time together, even if most of them may not instantiate yet. A task becomes a candidate of slot allocation when its belonging stage starts. Then it transits to the *waiting* phase, where it waits for being scheduled to an available machine slot. After that, this task runs. In the *input* phase, a task reads input data from either network or local storage. In most cases, the duration of this phase is dominated by network scheduling. The *computing* phase is CPU-intensive, thus can be considered as a relatively constant duration for a given computation workload running on a specific slot.

Which one is the last completed task of each stage is not predetermined. Allocating an available slot to one candidate task leads to increased waiting phases for all other candidates. Whenever a running task's input traffic is prioritized, input phases could be prolonged for all other tasks sharing the same network bottleneck. To sum up, scheduling of slots and networks largely dominates the variable part of each task's duration and, in turn, each stage's duration. Eventually, delays of every stage accumulate to a job's completion time.

Computing clusters are usually multi-tenant for utilization and cost-effective [5], [6]. An administrator's policy enforces slot fairness among users and/or jobs, hence which user/job is legitimate for the next slot is largely deterministic (e.g., First In First Out (FIFO) and Fair in Section 2.1). The design space includes: *which task to occupy which slot* (i.e., task placement), and *how to share the network among*

- *Chen Tian, Bingchuan Tian, Yang Zhao, Yuhang Zhou, Chenxu Wang, Wanchun Dou, and Guihai Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China. E-mail: {tianchen, douwc, gchen}@nju.edu.cn, bctian@smail.nju.edu.cn, {274131484, 1223870886, 806496096}@qq.com.*
- *Yi Wang is with the School of Modern Posts, Nanjing University of Posts and Telecommunications, Nanjing 210049, China. E-mail: 151485321@qq.com.*
- *Haoran Guan is with the School of Computer Science, University of Sydeny, Sydney, NSW 2006, Australia. E-mail: hgua5212@uni.sydney.edu.au.*

Manuscript received 18 November 2021; revised 28 April 2022; accepted 7 June 2022. Date of publication 17 June 2022; date of current version 23 August 2022. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2020B0101390001, in part by the National Natural Science Foundation of China under Grants 92067206, 62072228 and 61972222, in part by the Fundamental Research Funds for the Central Universities, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.
(Corresponding author: Yi Wang.)
Recommended for acceptance by S. Chandrasekaran.
Digital Object Identifier no. 10.1109/TPDS.2022.3182037

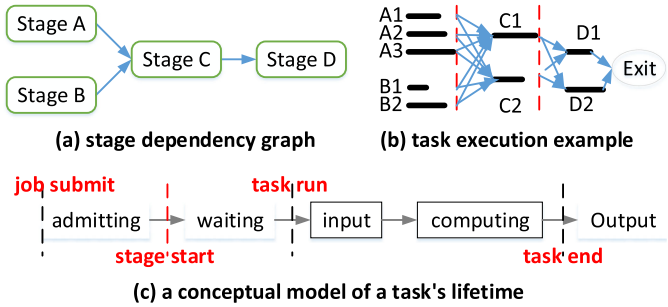


Fig. 1. Modeling a data-parallel job.

running tasks (i.e., network scheduling). These variables entangle when multiple jobs are executed in clusters and compete for slots.

Previous work does lead to improved performance with various strategies. *Locality-improve* strategy directly reduces network resource demand, where task placement considers data locality [7], [8], [9], [10], [11]. *Progress-require* strategy avoids slot resource waste, where task placement actively transfers slots from a legitimate job to other jobs when the legitimate job's tasks can not make progress if run on these slots [10], [12]. *Coflow-optimize* strategy shortens tasks' admitting phase, where network scheduling uses an abstraction of application semantics to shrink the time consumption of following stages' tasks in admitting phases [13], [14], [15], [16], [17] (Section 2.2). These task placement and network scheduling efforts are largely disjunctive. Task placement usually treats the network as a black box. Network scheduling targets network metrics. Without joint optimization, they can be sub-optimal and myopic in many scenarios (Section 2.3).

The key is to turn the black box into a white box to fully consider the performance impact of network scheduling in task placement for joint optimization. Our critical insight is that prioritized bandwidth allocation among tasks makes the network *predictable* and *efficient* [18], [19], [20], [21]. With this feature, joint scheduling can be transformed into a special *two-dimensional bin-packing problem*, where the first dimension is per-slot time, and the second dimension is different slots. However, this ideal abstraction may fail in some scenarios, where a single input phase task cannot exclusively utilize a machine's NIC even with network priority (Section 3).

We propose *p*redictable and *e*fficient *n*etwork *a*bstraction (Peña) as a compromise between ideal requirements and practical constraints. Peña approximates the ideal abstraction via *controlled but still prioritized concurrency* of input phase tasks' transmissions (Section 4).

Over this minimal yet power-enough abstraction, we propose PushBox to schedule data-parallel jobs in multi-tenant clusters. For a directed task execution graph, PushBox iteratively optimizes the current critical path. There are two challenges. First of all, the two-dimensional scheduler should guarantee a considerable performance improvement compared to only one-dimensional scheduling. Second and most importantly, the scheduler should respect administrators' fairness intent, which is largely ignored by previous work. To address all these challenges, we have designed our algorithms accordingly (Section 5).

We implement PushBox on Hadoop-3.1.1 (Section 6). PushBox is evaluated using both a 10-machine testbed and a trace-driven simulator. PushBox reduces the average/tail job completion time by 56.6%/51.9% compared with alternative approaches. We also evaluate PushBox under different scenarios. Results show that PushBox performs persistently well in almost all cases (Section 7). Discussions and future work are followed (Section 8).

2 Background and Motivation

In Section 2.1 we present the modeling of data-parallel jobs and targeted multi-tenant clusters. Related work is presented in Section 2.2. A bunch of toy examples demonstrates the opportunities for possible improvement (Section 2.3). To help presentation, we use Hadoop as the illustrative data-parallel framework due to its simple stage graph. A Hadoop job consists of a map stage of mappers, a reduce stage of reducers, and the communication among mappers and reducers (i.e., shuffle). The job completion time is equal to its reduce stage's completion time.

2.1 System Model

Data-Parallel Jobs. Shown in Fig. 1 is a general model of data-parallel jobs. For in-memory computing systems, SSD are widely used thus the bottleneck of disk IO can be neglected. Machine slots and networks are the main resource bottlenecks.

Network transmission in the sender-end usually does not consume slots. Output data are stored in memory (or fast SSD) and tasks terminate. Transmission in the receiver-end does consume slots (i.e., the input phase in Fig. 1). Readers may wonder how general the modeling of task lifetime in Fig. 1c is. For many tasks, although trivial pre-processing may exist within the input phase, main computing functions start only when all input data are collected (e.g., *terasort* reducers). Note that some tasks can begin main computing functions with partial data (e.g., *wordcount* reducers). If such a task's per-data-unit computing speed is faster than its data input rate, it is conceptually equivalent to having a 0-length computing phase. Otherwise, the task still has a computing phase. There are two types of dependencies among successive stages (proposed by Chowdhury *et al.* [15]).

- *Starts-After:* there exists an explicit barrier, where the following stage cannot start until its preceding stage has finished. For example, in synchronous machine learning, a new stage usually starts after the current stage finishes.
- *Finishes-Before:* the following stage can coexist with its preceding stage but it cannot finish first. Some dependencies with data transfer fall into this category. For example, the *slowstart* option in Hadoop enables a reduce stage to start when the ratio of completed mappers in the preceding map stage reaches a given threshold value.

We believe this modeling can be generalized to other stage-based pipeline types such as Spark's iterative optimizations common in machine learning.

Cluster Architecture. To make our analysis tractable, we assume homogeneous machines with equal resources. All

slots have equal resources, and a slot can support any task. Our examples use non-blocking networks. Non-blocking networks are characterized by the property that in the presence of a currently established connection between any pair of input/output, it will always be possible to establish a connection between any arbitrary unused pair of input/output. This model is attractive for its simplicity, and recent advances in data center network fabrics [22], [23] make these networks practical. Note that our approach does not assume or rely on non-blocking topology. Evaluations demonstrate that PushBox provides similar performance improvements in oversubscribed fabrics (Section 7). We defer discussion of readers' concerns to Section 8.

A scheduler continuously assigns available slots to waiting tasks from different jobs and recycles slots from completed tasks. Administrators specify fairness criteria for slot allocation. For example, Fair Scheduler enforces weighted-share among multiple users' job queues [24]. Each user queue can also specify either FIFO or Fair rules. In the FIFO mode, the scheduler can only allocate a slot to a following job's task if the currently running job is not disturbed. In the Fair mode, the available slots are weighted-shared among multiple concurrent jobs. The scheduler can set the Differentiated Services Code Point (DSCP) bit for each task's input flows to enforce network priorities. Network scheduling needs this capability to favor some flows over others (details in Section 2.2).

Scheduling Objectives and Outputs. Each task's input size can be directly get or roughly predicted [25], [26]. With the information of input size, a task's computing phase duration can be estimated from historical execution logs or by worst-case execution-time (WCET) code analysis [27]. Let each task's running time cost includes its durations of input and computing phases, Fig. 1b demonstrates a directed graph of task execution. An auxiliary *Exit* node is added and every task in the final stage points to it. A scheduler needs to minimize the time to finish all tasks in the graph.

Schedulers are activated in two modes. In the *slot-change* mode, there exist a number of candidate tasks in waiting phases. If a slot has just been released, it suggests new compute resources. If a slot's running task transits from input to computing phase, it suggests new network resources. As the fairness policy is already given, which job is legitimate for the next slot is deterministic. Scheduling decides which task of the legitimate job should be selected, and how network bandwidth is allocated to it. In the *task-arrival* mode, there exist a number of free slots. Then a batch of tasks just enters the waiting phases. Scheduling task placement in *task-arrival* mode is different since it has an additional step of deciding which slot should be placed on.

2.2 Related Work

There exists theoretical work in joint scheduling of network and computation [28], [29]. However, these works make impractical assumptions. For example, they require that all tasks' allocations are pre-determined at the start of the whole system. In reality, jobs arrive online and their tasks should be adaptively allocated to available slots.

Previous practical work has two separate working points: task placement and network scheduling. As a first step, we distill the underlying insights from them.

Task Placement. Locality-improve strategy reduces network resource demand. Being topology-aware, locality-oriented task placement directly reduces the burden for networks. On one hand, a localized task helps accelerate other running network-intensive tasks by reducing contention. On the other hand, it also has a chance to shrink the duration of its own input phase at the cost of a longer waiting phase [7]. Delay-scheduling and Quincy optimize data locality when placing mappers [7], [8]. Mantri instead optimizes data locality when placing reducers [9]. ShuffleWatcher performs shuffle-aware mapper and reducer placement in a multi-tenant cluster [10]. Corral performs joint input data and task placement to achieve better data locality based on workload prediction [11].

Progress-require strategy avoids slot resource waste. Some workplace tasks consider contemporary network load. If a task is already or would be, blocked in the input phase due to network contention, it may be better to donate this slot to another job of the same user, or even another user's jobs, so that other jobs could get the benefit. This is the intuition behind DynMR [12] and ShuffleWatcher [10]. DynMR monitors the progress of a reducer and backfills it with a mapper if data fetching is too slow [12]. ShuffleWatcher places a mapper instead of a reducer if the machine NIC utilization exceeds 75-100% [10].

Network Scheduling. Coflow-optimize strategy shortens tasks' admitting phases. The *coflow* abstraction considers application semantics. The *all-or-nothing* semantics means that all input flows to a stage must finish together. Previous network schedulers tries to minimize this make-span of data transfer between successive stages. Shorter admitting phases could be achieved for tasks belonging to the following stages. Some work try to minimize the average *coflow completion time* (CCT). Orchestra shows that even a simple FIFO discipline can significantly reduce the average CCT [30]. Barrat multiplexes multiple jobs' transfer to prevent head-of-line blocking [13]. Varys uses heuristics such as smallest-bottleneck-first and smallest-total-size-first [14]. Aalo is a non-clairvoyant scheduler that reduces CCT by following the Least-Attained-Service rule in a distributed manner [15]. Sincronia demonstrates that given a "right" ordering of coflows, average CCT within 4x of the optimal can be achieved [16]. Giroire *et al.* [31] divide the problem of scheduling network tasks into two subproblems: choosing the placement policy that minimizes the network and computational overhead and scheduling the tasks under the above placement policy. This scheduling algorithm is optimal on simple MapReduce workflows.

Summary. State-of-the-art task placement and network scheduling efforts do lead to improved performance. While they are largely disjunctive. Task placement work usually treats the network as a black box. They thus focus on reducing resource demand or waste. Network scheduling work targets network metrics such as CCT. As far as we know, NEAT is the only task placement approach with explicit network prediction [32]. Given accurate network status and a network scheduling policy (i.e., First-Come-First-Serve, Least-Attained-Service and Fair), a centralized scheduler uses the predicted transmission completion time to place tasks that can accomplish their flows/coflows at the earliest time. NEAT only handles the *task-arrival* mode, where slots are always sufficient for

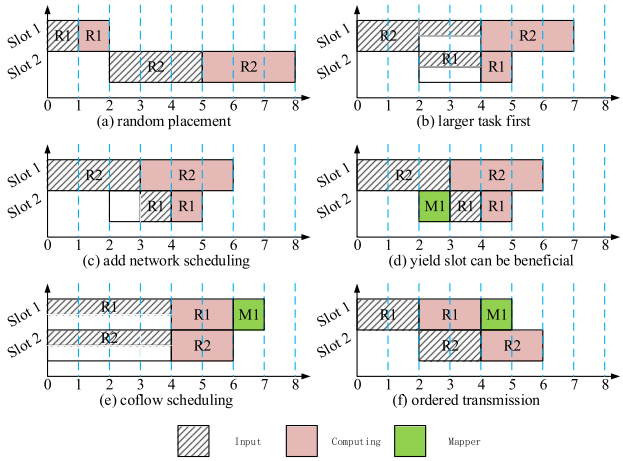


Fig. 2. Performance improvement opportunities.

placing every task. It targets network metrics such as *flow completion time* (FCT) or CCT. NEAT takes any existing network scheduling policy for granted as an algorithm input. It does not consider joint optimization of computing and network.

2.3 Our Observations

Via several toy examples, we present our observations that are common in practice where existing approaches are sub-optimal and myopic. All examples are in slot-change mode.

Case 1: Task Placement Can Exploit Intra-Stage Task Heterogeneity. Current work largely ignore the native heterogeneity among tasks even in the same stage. Complex analytics in data-parallel framework's programming practice increases the demand for user-defined operations (UDOs). Without sophisticated custom partitioning functions or heavy-weight load-balance mechanisms, data skewness is inevitable [33], [34]. In the same stage, one task may have much more input data than another task. With the same computing function, its computing phase could be also much longer.

Consider a single Hadoop job example in Fig. 2a. There is a single job queue with one job A. Job A's map stage has already finished at time 0. There are two reducers R_1/R_2. R_1 has 1 unit of shuffle data to be collected and 1 unit of computation time. R_2's data and computation both triple that of R_1. A machine with two slots $S1/S2$ can receive at most 1 unit of data in each time unit. $S1$ is available at time 0 and $S2$ is free later at time 2. In this figure, we use textured/stuffed rectangles to denote task input/computing phases.

At time 0, current task schedulers may randomly select R_1 for $S1$. Then at time 2, R_2 can be scheduled to either $S1$ or $S2$. Job A's completion time is 8. This is far from optimal. Suppose R_2 is scheduled to $S1$ at time 0 in Fig. 2b. Then R_1 is scheduled to $S2$ at time 2. Between time 2 and 4, they equally share the bandwidth. Job A's completion time can be reduced to 7.

Case 2: Network Scheduling can Further Exploit Intra-Stage Task Heterogeneity. Same with those task placement approaches, existing network schedulers also never consider the length of each task's computing phase. In this example, R_2's computing phase is much longer. An improvement is to parallelize network and computation as much as possible. As shown in Fig. 2c, if R_2's flows can be prioritized over R_1's flows, R_2's and job A's completion time can be further reduced to 6.

Case 3: Task Placement may be Myopic to Waste Slots. Now let's extend the above example to a single-queue multi-job scenario. We assume the slot policy is FIFO. There is an additional job B, which only has a single mapper task M_1 with 1 unit of computation time. Its submission time is 2+δ, where δ is a negligibly small value.

Most schedulers (e.g., ShuffleWatcher) always place a task if there exist available slots. Hence R_1 would occupy $S2$ at time 2 regardless of network scheduling. Following the network scheduling in Fig. 2c, R_1 would actually block without progress for 1 time slot. At time 2+δ, job B is committed but all slots are occupied. M_1 has to wait until $S2$ is released at time 5. Job B's completion time is 6-(2+δ) = 4.

The key observation is that, if we have already known a priori revelation that a task would be blocked by the network, we can choose to not place it at all and keep the slot empty. Then $S2$ is not allocated at time 2. As mentioned above, the FIFO scheduler can allocate a slot to a following job's task if the currently running job is not disturbed. As shown in Fig. 2d, then M_1 can be allocated at time 2+δ. Job B's completion time is only 1 unit.

Case 4: Network Scheduling may Waste Slots Unnecessarily. Note that network scheduling alone could waste slots regardless of data skewness and task placement. We modify the example to evenly split the input data between R_1 and R_2 assuming a perfect data balance [33]. Now they both have 2 units of shuffle data and 2 units of computation time. We also assume both slots are available at time 0.

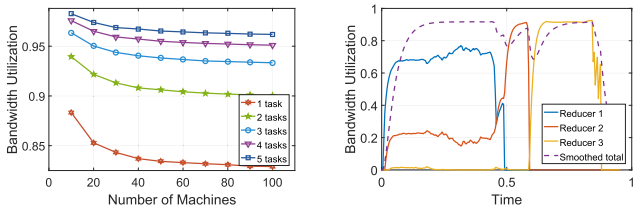
A widely-accepted mantra is to finish all flows simultaneously within a coflow [14], [15], [16], [30]. The example in Fig. 2e follows the rule. In this case, both tasks finish at time 6 and M_1 can be allocated. As a comparison, if we let R_1's flows have priority, then R_1 can be finished at time 4 without impacting the coflow finish time (Fig. 2f). Then M_1 can be scheduled at time 4 and job B's completion time is 2 units earlier. Similar results have been reported recently [35].

Summary. It is clear that joint optimization of compute and network can accelerate job completion. While as mentioned above, optimal joint scheduling of compute and network is NP-hard.

3 Overview

Intuition. Our key insight is that prioritized bandwidth allocation among tasks makes the network both *predictable* and *efficient*. With this feature, joint scheduling of compute and network can be transformed into a special *two-dimensional bin-packing problem*, where the first dimension is per-slot time, and the second dimension is different slots.

Let's revisit the hidden network assumptions of toy examples: 1) tasks can be prioritized, and 2) the highest priority task can exclusively use full NIC bandwidth. Thus, the duration for each task's input phase can be accurately estimated. The scheduler can choose to place larger task R_2 first and harvest the parallelism between R_2's computing and R_1's input phases (i.e., Case 2 in Section 2.3). It can also choose to be far-seeing instead of myopic by keeping the slot free for the moment (i.e., Case 3 in Section 2.3). For



(a) Utilization v.s. tasks (b) Emulate ideal Peña

Fig. 3. Approximate the ideal network abstraction.

already-placed tasks, prioritized network scheduling can save slots for other jobs without compromising a target job's progress (i.e., Case 4 in Section 2.3).

Now a scheduler is relieved from fine-grained bandwidth allocation. Intuitively, a task can be represented by a box with two fixed-length segments (i.e., network and compute). The scheduler then packs tasks in a job's graph one-by-one to slot-time two-dimensional bins.

Challenge. The second assumption of examples is not practical in many scenarios: there is no guarantee that a single input phase task can exclusively utilize a machine's NIC even with network priority. An input phase task may retrieve data from all machines of preceding stages. It initiates multiple connections simultaneously to exploit available bandwidth. Usually, the receiver-end is the network bottleneck. While in some cases congestion could exist in sender-end. For example, a non-local Hadoop mapper needs to read input data from 1 HDFS copy in another machine. What's more, modern datacenter networks usually have bandwidth fan-in (a term that defines the maximum number of digital inputs that a single logic gate can accept) between leaf and spine switches [23]. Input flows could be congested at the network core than at the receiver-end.

Scheduling based on the ideal abstraction could be neither predictable nor efficient. For Case 3 in Section 2.3, if R_2 cannot fully utilize the bandwidth as predicted, it is a waste of network resources to delay placing R_1.

4 A Practical Abstraction

*P*redictable and *e*fficient *n*etwork *a*bstraction (Peña) is a compromise between ideal requirements and practical constraints. In Spanish, Peña means "rocky summit". PushBox is the first attempt to define a dedicated network abstraction to jointly schedule compute and network for data-parallel jobs.

Observation. Running multiple connections concurrently can help improve utilization. We simulate bandwidth utilization with a number of machines in a 10 Gbps non-blocking datacenter network. In each machine, there are m input phase tasks with random settings of network priority bits. Each task establishes 5 random connections to pull data from remote machines, which is the default setting in Hadoop. As shown in Fig. 3a, only 3 tasks can already use about 95% bandwidth. The additional throughput gain brought by one more task per machine decreases exponentially. We can expect that in an oversubscribed network, the utilization of bottlenecked core switches could be even higher. It is safe to consider a network reaches a congested status, once the number of input phase tasks reaches a threshold.

Controlled Concurrency. Our solution is *controlled but still prioritized concurrency* of input phase tasks' transmissions. We use a small number m to control the number of input phase tasks. The choice of m is a trade-off between slot and network resources. A too-small m may lead to low network utilization, while a too-large m may waste slots. The value of m is related to network topology and NIC link speed. Usually, a sweep of several values can get a reasonable value range of it. We use $m = 3$ for our non-blocking 10 Gbps testbed, and leave the analysis of optimal m to future work.

Note that m is a recommended lower bound to the scheduler. A scheduler can decide to temporarily increase the number of input phase tasks more than m when it is beneficial or necessary (Section 5.2). In each machine, network priority is supported among input phase tasks by assigning different DSCP bits independently.

Being Efficient. Controlled concurrency guarantees network utilization. Prioritized concurrency guarantees that task transmissions are approximately ordered if they compete for the same network bottleneck. Let *header task* refers to the input phase task with the highest priority in a machine, and let *tail tasks* refer to others. With the highest priority in the whole network, a header task is unlikely to be bottlenecked by any tail task. Its competition comes from other header tasks in other machines. With higher bandwidth, high-priority tasks finish faster to release their slots. Although not optimal, it is already much more efficient (in terms of slot resource) than the original flow-level fair sharing or coflow-level scheduling.

A measure experiment is shown in Fig. 3b, where three Hadoop reducers share the same machine. They have the same input sizes and the network priorities are $R_1 > R_2 > R_3$. There exist other 8 machines with running reducers. Compared with the ideal abstraction, network bandwidth is still almost fully utilized but not perfectly exclusive due to mapper end competition. The completion times of reducers are ordered.

Being Predictable. Given the nature of controlled concurrency, the prediction of congestion status is accurate. The problem is how to estimate the input phase duration of a task. As shown in Fig. 3b, it is hard for a tail task (e.g., R_2) whose duration is dominated by all running tasks with higher priorities.

We take a step back and only use the measured total receiving rate of the machine NIC in our scheduling. The receiving rate can be estimated as received bytes divided by elapsed time. The obtained "effective" machine bandwidth is smoothed using an exponential moving average algorithm. As demonstrated in Fig. 3b, this rate is relatively stable compared with the per-task rate. Thus, the remaining duration of all running tasks' transmissions, together with a candidate task's duration in this machine, can be roughly predicted as the number of remaining bytes in this task divided by the estimated receiving rate. With this time, we can better predict the duration of the input phase of the task for *slot selection* (Section 5.1).

Note that for large oversubscribed networks, the intra-rack transfer is not accounted in this calculation. In this scenario, the rack-level data source is also considered as local. Intra-rack transmissions are given a reserved lowest network priority to exploit all remaining bandwidth. An example is a mapper with rack-locality.

5 PushBox Scheduler

Over the minimal yet power-enough abstraction Peña, we propose PushBox to schedule data-parallel jobs in multi-tenant clusters. PushBox scheduling focuses on the placement ordering of tasks. A well-designed heuristic algorithm should obtain near-optimal job completion time.

Critical-Path Oriented Scheduling. PushBox gets insights from the Dynamic Critical Path (DCP) algorithm [36], where the critical path is defined as the current longest path from a candidate task to the exit node (Fig. 1b). PushBox iteratively optimizes the critical path during the scheduling process. The intuition is that the critical path defines the lower bound of job completion time.

The stage-based pipeline model of data-parallel jobs presents a unique opportunity to find the critical path. From a stage-centric view, there is always a running stage in the critical path as a whole. Scheduling problem is transformed to select the critical task from current critical stage and assign network priority to the task.

Challenges. To minimize job completion time, the scheduler needs to consider multiple factors besides critical path of the legitimate job. We use the word *legitimate* to emphasize that the scheduler may transfer slots from a legitimate job to other jobs, either due to locality-improving or progress-require strategies. This job/task is called the *beneficiary* job/task.

First, the two-dimensional scheduler should guarantee a considerable performance improvement compared to only one-dimensional scheduling. To achieve near-optimal performance, we embraced the existing wisdom of related work, including task placement and network scheduling. For example, locality-improve is a trade-off between longer waiting phases and possibly shorter input phases for the current legitimate job tasks. Without care, it may hurt job completion time. We incorporated these strategies when designing building blocks (Section 5.1) and achieved better performance than the original through a careful design.

Second, the scheduler needs to respect administrators' fairness intent. Administrators do have an implicit "assumption" about the underlying network in actual scheduling. They expect that bandwidth is equally shared among running tasks [37]. If the number of running tasks follows Fair or FIFO policies, the network may also approximate these policies. However, this "assumption" could be compromised by various optimization strategies. So far this important issue is largely ignored by related work. We add several adaptions to our algorithms (Section 5.2).

Putting them together, we present the pseudo-codes of scheduling in two modes (Section 5.3).

5.1 Building Blocks

For each invocation of the scheduler, it first gets an ordered job list according to a given multi-queue scheduling policy. For both slot-change and task-arrival modes, scheduling starts with *task selection* from a legitimate job. Task-arrival mode has an additional step of *slot selection*. The next step is *controlled concurrency* to judge whether the chosen task can be placed. If the task passes the test, the task is actually placed with a network priority setting. Fail any step, the slot should be transferred to the next job in the list.

Algorithm 1. Building blocks

```
1  function selectTask(tasks, machine)
2    if tasks have oppotunities of locality then
3      selectedTask ← select one task according to existing
       locality polices;
4      return selectedTask;
5    else
6      sort tasks in descending order by input size;
7      return the first task;
8  end
9  function canBePlaced(task, machine)
10   if machine.#inputPhaseTask < threshold then
11     return true;
12   else
13     if task.job.#coflow < machine.minCoflow then
14       if policy = Fair then
15         return true;
16       else
17         return false;
18     else
19       return false;
20 end
21 function notHinderTask(task,machine)
22   if task.compuingTime < machine.dataTransferTime then
23     return true;
24   return false;
25 end
```

Task Selection. Consistent with locality-improving strategy, task selection first branches if the current stage has an opportunity of locality (line 2 of Algorithm 1). In PushBox, we use existing locality policies but only for a limited range of tasks (lines 3-4 of Algorithm 1). For example, the first stage of a job usually reads input data from distributed file systems (e.g., mapper tasks in Hadoop). For each task, the number of possible data source machines is limited (e.g., usually 3 in HDFS). Localizing such a task can eliminate all network traffic (machine-local) or, all cross-rack traffic (rack-local). PushBox does not consider a task with a large-number of data sources (e.g., reducer tasks in Hadoop). In a cluster with hundreds of racks of machines, the benefit of localizing such a task is questionable while it prolongs the waiting phase of this critical task.

To handle the trade-off between the waiting phase and input phase, PushBox is compatible with the Delay-scheduling approach. Delay-scheduling adds a mechanism to limit the maximum number of slot transfers [7]. It gives a slot to a legitimate job after a certain number of skips. Thus the *returnTask* could be a local task without input phase, or a task with remote input data, or *null* (i.e., there is no local opportunity while the skip count is not reached yet).

For the other branch, we need to find the critical path. The exact estimation of task durations is actually not required in this part. The observation is that all tasks in a stage share the same computing function [2]. The larger the input data, the longer the computing phase. As a result, PushBox sorts all waiting tasks in the critical stage in descending order of their input sizes and selects the largest task as the candidate (lines 5-7 of Algorithm 1).

Slot Selection. Machines with free slots are sorted according to their remaining total network duration, i.e., remaining

bytes divided by estimated machine receiving rate. In the bin-packing context, it is equivalent to pack the chosen task as early as possible. The corresponding codes are shown in Algorithm 3.

Controlled Concurrency. For a given slot, the scheduler judges whether a given machine's network is already congested (lines 9-10 of Algorithm 1). Otherwise, a new task with input data requirements can be placed.

Flow Priority. When a new input phase task is placed, or a running task's input phase finishes, a machine re-orders the flow priorities of current input phase tasks. Here PushBox should respect administrator intent. For tasks inside the same job, their priorities are assigned according to their placement ordering. For tasks among different jobs, Fair and FIFO are different.

Suppose a smaller job A and a larger job B with the same IO-size/computing-time ratio. Resort to the original Fair intent, administrators hope that job A is not head-blocked by job B if B arrives a little earlier. By fair-sharing both slots and network, job A should accomplish earlier. Here we use the coflow-optimize strategy. Different tasks are sorted in ascending order by their belonging coflows' sizes (i.e., smallest-coflow-first). Readers may wonder why not also share the network according to Fair policy? The intuition is ideal, if the network is not consistently congested and B's input size is large enough, job B's input phase completion time would not be hurt. At the same time, we can reduce the completion time of small jobs.

The exception is that one job queue's policy is FIFO. This FIFO job queue must have a head job. Only tasks of this head job follow the smallest-coflow-first rule in a multi-queue scenario. For other-than-head jobs' tasks in a FIFO queue, a reserved lowest priority is given to them. Note that concurrency is a quite small number, hence each machine usually uses only a small number of priority values.

This flow priority setting is different from strict bin-packing rules such as putting a new box at the end of the time dimension. A new task's flows might be promoted, while a running task's flows might be preempted. This is natural in online scheduling. Scheduling only gives an estimation of finish time. Traffic preemption from new jobs is permitted as long as it is consistent with administrator intent.

5.2 Intent-Oriented Adaption

Not-Hinder Test of Beneficiary Tasks.

With progress-require strategy, the necessary and sufficient condition for yielding a slot is that the network is already congested. The beneficiary task should only contain the computing phase. But without care, the beneficiary task might deteriorate network utilization in certain scenarios.

We demonstrate a Hadoop example in Fig. 4. Job A has 4 reducers R_1 to R_4. Job B/C each has only one mapper M_1/M_2. The job scheduling sequence is A, B, and C. There are 4 slots in a machine, and the network concurrency limitation $m = 3$. With Peña abstraction, the scheduler puts R_1 to R_3 to slots. It judges that R_4 can not make progress because of the network saturation. In this case, the slot is yielded to following job B's mapper M_1, as shown in Fig. 4a. However, M_1 runs too long. When input phases of R_1, R_2, R_3 are all finished, the network becomes idle. Since all slots are already occupied, R_4 can not be allocated. Thus an unsuitable

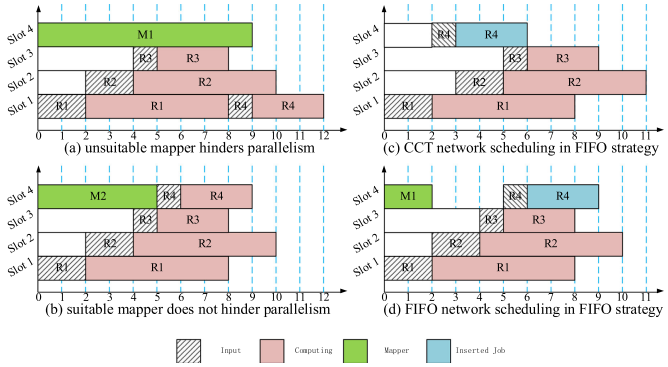


Fig. 4. Avoid conflict to intent.

beneficiary mapper may hurt network utilization. This is the conflict between respect intent and slot transfer. Other progress-require strategy approaches (i.e., DynMR and ShuffleWatcher) actually share the same issue.

To prevent this situation, PushBox adds a test for possible beneficiary tasks (function `notHinderTask` in Algorithm 1). Given the current NIC utilization and remaining bytes of all tasks' input phases, the overall remaining network duration (i.e., *dataTransferTime*) can be estimated. If the chosen task's *computingTime* is less than the machine's *dataTransferTime*, PushBox considers it as a suitable task and place it. Fig. 4b shows that a suitable mapper does not hurt network.

Wildcard in Concurrency Control. Consider the administrator intent: allocating a slot also means a share of the network. Without care, concurrency control may negate this intent. For example, suppose a machine already uses 3 slots for a larger job's input-phase tasks. According to the fairness policy, a smaller job is the next legitimate job and it progresses to the reducer stage. However, this machine is saturated so the reducer tasks of the smaller job cannot be placed. In other words, the job with smaller coflow is blocked by our concurrency control. This is a conflict between concurrency control and respect administrator intent.

We set a wild-card rule for Fair policy. The details are shown in function `canBePlaced` (lines 11-14 of Algorithm 1). If the size of the new coflow is smaller even than the smallest coflow in a machine, we let the smaller job has a chance to compete for bandwidth. Otherwise, we still follow the concurrency control rule. We do not prioritize a large coflow, because its finish time should be later in fair sharing scenarios.

Note that for FIFO mode, we do not give wildcards. An example is shown in Fig. 4c, a smaller job of R_4 is inserted. If we give it the wild card, the original job is affected. This is a conflict because the administrator prefers Fig. 4d (lines 15-16 of Algorithm 1).

5.3 Putting Them All Together

Algorithm 2: Slot-Change Mode. The *slotYield* flag represents whether slot transfer from the legitimate job/tasks to beneficiary job/tasks has happened, and we initialize it to 0. PushBox traverses the job list to pick the next candidate task by calling `selectTask` (line 4). Without a returned task, the slot should be transferred to the next job (lines 5-7). If the task is local, the job has the highest allocation opportunity. If this task belongs to the legitimate user/job, we can place this task (lines 9-10). Otherwise, it should pass the test to not

hurt the bandwidth (lines 11-13). If the task is a remote task with an input phase, PushBox judges whether the task can be placed on this machine (lines 15-16). The *slotYield* flag is set to 1 if the task can not be placed (lines 17-18).

Algorithm 2. Slot-Change Mode

```
1  Jobs[ ] ← multi-queue job scheduling order;
2  slotYield ← 0;
3  foreach job in Jobs do
4    chosenTask ← selectTask(job.tasks, slot.machine);
5    if chosenTask = null then
6      slotYield ← 1;
7      continue;
8    if chosenTask is local then
9      if slotYield = 0 then
10       return chosenTask;
11     else
12       if notHinderTask(chosenTask, slot.machine)) then
13         return chosenTask;
14   else
15     if canBePlaced(chosenTask, slot.machine) then
16       return chosenTask;
17     else
18       slotYield ← 1;
19 return null;
```

Algorithm 3. Task-Arrival Mode

```
1  taskMachinePair ← ∅;
2  placed ← 1;
3  while placed do
4    placed ← 0;
5    machines[ ] ← sort in ascending order by
     dataTransferTime;
6    foreach machine in machines do
7      if machine.#freeSlot = 0 then
8        continue;
9      chosenTask ← selectTask(tasks, machine);
10     if chosenTask = null then
11       continue;
12     if chosenTask is local then
13       Jobs[ ] ← multiq-queue job scheduling order;
14       if chosenTask.job = Jobs[0] then
15         taskMachinePair.add(chosenTask, machine);
16         placed ← 1;
17         break;
18       else
19         if notHinderTask(chosenTask, machine) then
20           taskMachinePair.add(chosenTask, machine);
21           placed ← 1;
22           break;
23     else
24       if canBePlaced(chosenTask, machine) then
25         taskMachinePair.add(chosenTask, machine);
26         update(machine.dataTransferSize);
27         tasks.remove(chosenTask);
28         placed ← 1;
29         break;
```

Algorithm 3: Task-Arrival Mode. In this mode, PushBox is matching a task list with a slot list. $taskMachinePair$ records matched task-slot pairs and *placed* controls in the loop. At the beginning of each loop, PushBox sorts all machines with free slots in ascending order by `dataTransferTime` (line 5). Then we traverse all machines and pick up a task (lines 9). Suppose the task is local. Before allocation, we must verify this job have the highest allocation priority (lines 14-17). Otherwise, we only allocate suitable tasks (lines 18-22). If the task has an input phase, PushBox judges whether this task can be placed on a machine (lines 23-29).

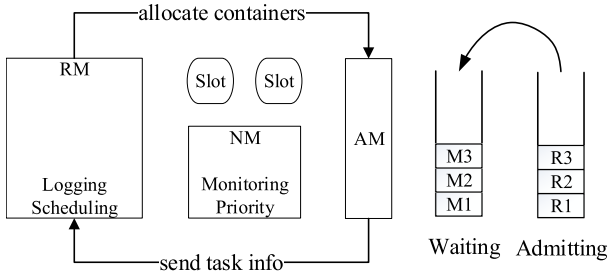


Fig. 5. Hadoop implementation.

6 PushBox in Testbed

6.1 Implementation

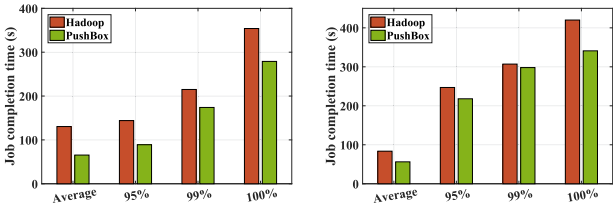
We have implemented PushBox in Hadoop 3.3.1 with the YARN manager [6]. We modify Resource Manager (RM), Node Manager (NM) and Application Master (AM) with more than 400 lines of source code. The architecture is illustrated in Fig. 5.

AM is an independent management process per job, which periodically queries containers for its waiting tasks from RM. RM runs PushBox scheduler, which uses delay scheduling [7] as a framework. We implemented the logging module in RM to record and analyze the relationship between input size and computing time for each kind of application, thus we can predict the task computing phase duration. When enabling the slowstart option, PushBox can still estimate the order even with partially finished a map stage, because the input data distribution of a reducer is similar in all mappers for most Hadoop applications. After slot allocation, the task together with metadata is sent to the corresponding machine. NM locates in slave machines to manage local slots. We implement the network monitoring model in NM. Reducers use HTTP to pull input data from remote machines. We modified the underlaying socket implementation to give each flow a priority by setting the DSCP field in the IP header.

6.2 Performance Evaluation

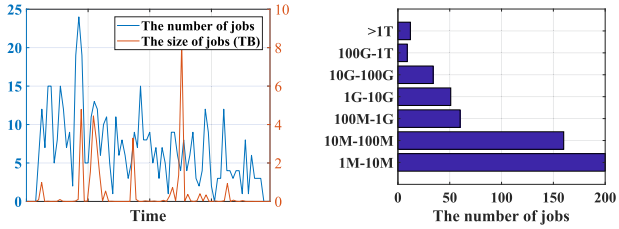
Setup. We evaluate our Hadoop implementation in a testbed where 10 servers connect to a 32-port Arista switch via 10 Gbps links. A master machine runs RM, and 9 slave machines run NM. AM is a normal process running in a dedicated container. Each server is a DELL PowerEdge R730, equipped with two 12-core Intel Xeon E5-2650V4 CPU, 256 GB RAM, and a 1 TB HDD. The operating system is Ubuntu Server 14.04 with Linux 4.4.0 kernel. We map RAM to disk with the `tmpfs` command to store all data in memory.

*Workload.*We use terasort as background IO-intensive jobs and other applications are foreground computation-intensive jobs. Workload 1 has two terasort (i.e., 300 GB and



(a) workload 1 (b) workload 2

Fig. 6. Testbed results.



(a) Temporal pattern (b) Size pattern

Fig. 7. Traffic patterns in Facebook logs.

100 Gb) and 28 small query jobs. Workload 2 is mixed by 30 PUMA benchmark applications [38] including a 300 GB terasort, an 100 GB terasort, a 50 GB wordcount, three 5 GB grep, three 1 GB grep, ten 100 MB teragen, and ten 10 MB teragen.

Hadoop Parameters. We allocate the same CPU (1 core) and memory (1.5 GB) container resources for different tasks (e.g., AMs, mappers, and reducers). The rest of the memory is used to store data. A machine can run at most 10 slots. The HDFS block size is 1 GB, and there are 3 replicas for each block. There is a single job queue with the Fair policy. For terasort, the number of sampling is reduced to emulate data skew in intermediate output.

Performance. We evaluate PushBox in two workloads. As shown in Fig. 6, PushBox performs better than Hadoop in both workloads. As mentioned above, PushBox saves slots aggressively thus some small jobs finish faster. Compared with Hadoop, PushBox reduces the average and tail job completion time by 49.8% and 21.1%, respectively in workload 1, and by 32.6% and 18.8%, respectively in workload 2.

7 Large Scale Simulation

7.1 Methodology

We develop an event-driven flow-level simulator to comprehensively evaluate PushBox in different scenarios. Our workload is generated from a collected Facebook log [39]. This log is widely accepted as a benchmark for coflow-related scheduling work [14], [15], [40], [41], [42], [43]. Bursty traffic patterns can be observed in Fig. 7. We do not modify jobs' arrival time and sizes in the log, and there are over 30 TB shuffle traffics in total.

The Facebook log only contains shuffle traffic of these jobs. Based on measured production data [44], [45], we generate the settings of mappers according to shuffle size and input/output data ratios. In default settings, time spent on the input phase and the computing phase is predictable. Unless otherwise specified, we use these settings by default.

Metrics. We use the average job completion time as our main metric. Besides, we take the 95/99/100th percentile job completion time into consideration. To validate the effectiveness of each component, we replace each design component with a trivial solution or disable it. Then we compare PushBox with state-of-the-art algorithms, including NEAT, ShuffleWatcher, and the default scheduler of Hadoop. Here we use the CCT^{TCF} version of NEAT with sequential heuristics and the same network priority levels for a fair comparison.

Default Settings. In our simulator, we abstract the topology similar to our testbed except that the number of machines is 20 and the number of slots per machine is 20. We evaluate a blocking network topology with 64 machines in Section 7.3. This type of network cannot realize all possible connections between inputs and outputs because an existing connection in the network blocks a connection between one free input to another free output. Unless other specified, we use these as default settings. Every data point is an average of multiple runs.

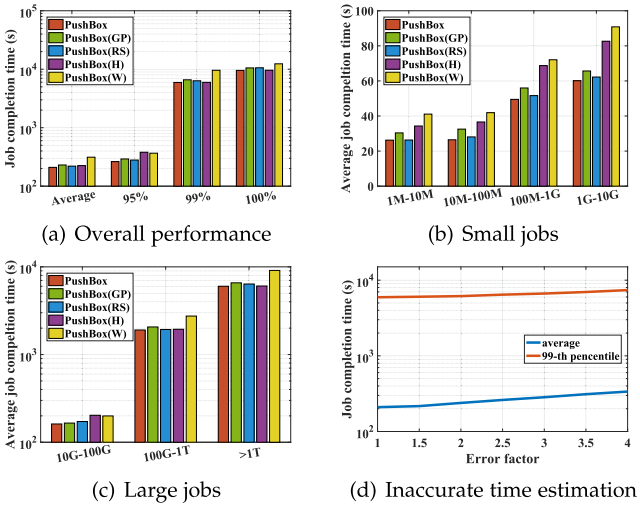
7.2 Dissect PushBox

Impacts of Components. First, to demonstrate the benefits of our network abstraction, we replace it with a global policy that follows the smallest-coflow-first principle. Here we partition all coflows into several bins according to their size, and coflows in the same bin have the same network priority. Results are marked with a suffix `GP`, which means Global Priority. Then, we use suffix `RS` to represent the scenarios that critical-path-oriented task selection is replaced with a Random task Selection. Similarly, we use suffix `H` to represent the scenarios that not-Hinder policy is disabled and use `W` to represent the scenarios that Wild-card in concurrency control is disabled.

Results are shown in Fig. 8. It illustrates that each component in PushBox is indispensable. Generally, without one component, the average job completion time grows by 9.9% (GP), 5.0%(RS), 7.1%(H) and 49.3%(W) respectively, and the 95% tail job completion time has various degrees of growth (11.1%(GP), 6.5%(RS), 44.8%(H) and 39.4%(W)) (Fig. 8a). To investigate the impact in detail, we partition all jobs into 7 bins according to their sizes. As shown in Figs. 8b and 8c, such replacements hurt the performance evenly for jobs in each bin. For small jobs, such replacements grows average job completion time by 15.3%(GP), 3.5%(RS), 36.3%(H) and 52.9%(W) respectively. For large jobs, such replacements grow average job completion time by 6.7%(GP), 4.7%(RS), 9.4%(H) and 39.7%(W) respectively.

Impacts of Information Inaccuracy. Another important problem is, does PushBox suffer significantly if Peña gives estimation of inaccurate input phase duration? Here we introduce a random estimation error E, and multiplying task input phase duration with it to emulate the estimation error in real systems. Assuming a random variable $U \in (0, 1)$ is sampled from a uniform distribution, thus $E = f^{2U-1}$, where $f \in [1, +\infty)$ is the error factor, which means the estimation error can be as large as f times.

Results (Fig. 8d) show that both average and tail job completion time increase as the estimation error grows. If the estimation error grows by $2\times$, the average job completion time increases by 13.8%, and tail job completion time increases by 3.5%. If the estimation error grows by $4\times$, the



(a) Overall performance (b) Small jobs

(c) Large jobs (d) Inaccurate time estimation

Fig. 8. Every component is indispensable.

average and tail job completion time can increase by 60.7% and 24.3%. Average job completion time is more sensitive because the wrong estimation of durations affects task order, while the tail job completion time is dominated by throughput. To sum up, PushBox is sensitive to the estimation error, but reasonable inaccuracy cannot severely affect the performance.

7.3 Comparison Results

We compare PushBox with NEAT and ShuffleWatcher, which are known as state-of-the-art algorithms. Besides, we use default scheduling algorithms in Hadoop as the baseline. We evaluate these algorithms in many different scenarios, such as job queue settings, oversubscribed ratios, cluster scales, the number of slots per host, workloads, and so on.

Job Queue Settings. As shown in Fig. 9, we evaluate PushBox with Fair, FIFO, and Multi-queue (one Fair queue and one FIFO queue) settings. Fig. 9a shows that PushBox and ShuffleWatcher reduce all job completion times, and NEAT only reduces smaller jobs' completion time. PushBox performs best all the time. Compared with Hadoop, ShuffleWatcher and NEAT, PushBox reduces average job completion time by 57.5%, 40.0%, and 52.1%, respectively. Meanwhile, PushBox reduces 95% tail job completion time by 87.4%, 39.5% and 65.3%, respectively.

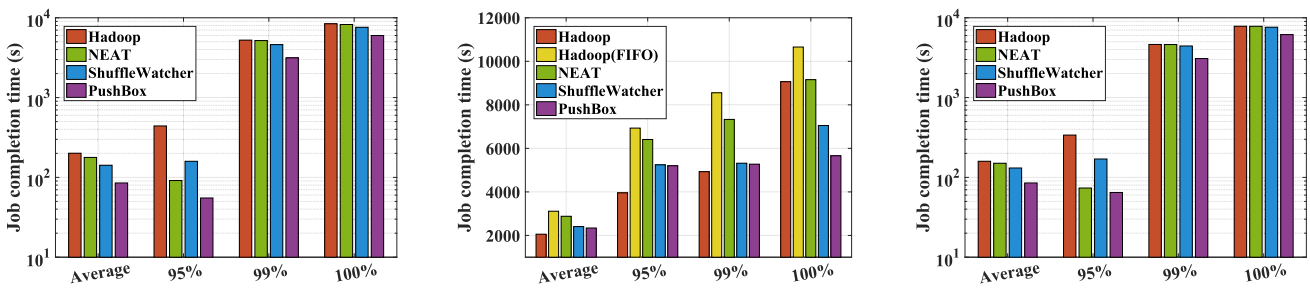
Fig. 9b shows that Hadoop is the best algorithm. The reason is that it fails to comply with the FIFO semantic. A later and smaller job can finish first because of network fair-sharing. Actually, original ShuffleWatcher and NEAT also are better than PushBox by not respecting FIFO intent in networking. We implement a network-FIFO version of Hadoop, marked as Hadoop(FIFO). We also modify ShuffleWatcher and NEAT to follow the FIFO rule. It is clear that now PushBox has the best performance when all algorithms keep the FIFO semantic. ShuffleWatcher also performs well, because it has the slot transfer feature. With FIFO semantic, the optimization room for network scheduling is very small. Compared with Hadoop(FIFO), ShuffleWatcher, and NEAT, PushBox reduces the average job completion time by 24.7%, 2.7% and 18.7%, and tail job completion time by 46.8%, 19.7% and 38.1%, respectively.

We mix a FIFO queue and a Fair queue to constitute the multi-queue setting. All small jobs (<100 MB) reside in the FIFO queue to prevent one big job from blocking small jobs. Fig. 9c shows the average job completion time of Hadoop, ShuffleWatcher and NEAT decrease but PushBox performs even better. Compared with Hadoop, ShuffleWatcher and NEAT, PushBox reduces the average job completion time by 21.4%, 19.2% and 21.2% and 99% tail job completion time by 33.4%, 30.3% and 33.2%, respectively.

Oversubscribed Ratios. We evaluate PushBox with different bandwidth fan-in ratios in an oversubscribed network. Results are shown in Fig. 10. Here we have 4 clusters and 16 machines in each cluster. The topology is layered with 64 machines. The fan-in factor varies from 1:1 to 6:1, which means the total intra-cluster bandwidth is 1-6 times larger than the total inter-cluster bandwidth.

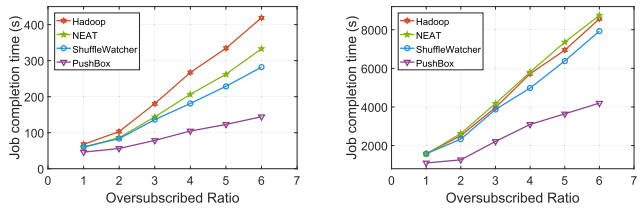
The average and tail job completion time of all schedulers becomes larger as the oversubscribed ratio increases, because of the bandwidth limitation of the core network. Results show that PushBox performs well in oversubscribed topologies. Although PushBox is a joint optimization of computing and network, its main benefits come from network scheduling. Usually, the more congested network is, the better the performance of PushBox. Compared with Hadoop, ShuffleWatcher, and NEAT, PushBox reduces the average job completion time by 31.7%-65.5%, 23.7%-48.8% and 21.8%-56.6%, and tail job completion time by 30.1%-50.9%, 30.3%-47.0%, and 29.4%-51.9%, respectively.

What if More Resources. Without changing workload, we evaluate PushBox in different cluster scales from 10 to 110 machines. As shown in Fig. 11, as the cluster scale grows larger, the performance of all algorithms becomes better and the differences among them become smaller. With more hosts, there are more network and slot resources. Both network and slot bottlenecks gradually diminish. Compared



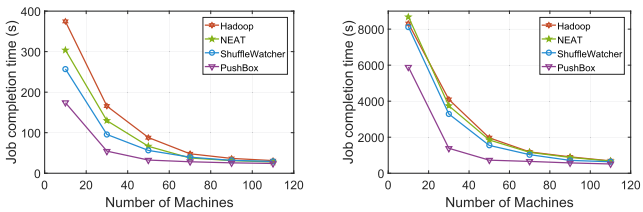
(a) Single Fair queue (b) Single FIFO queue (c) Multi-queue one Fair one FIFO

Fig. 9. Performance in different settings.



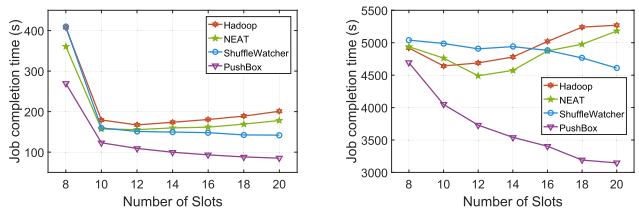
(a) Average completion time (b) 99% tail completion time

Fig. 10. Impact of oversubscribed ratio.



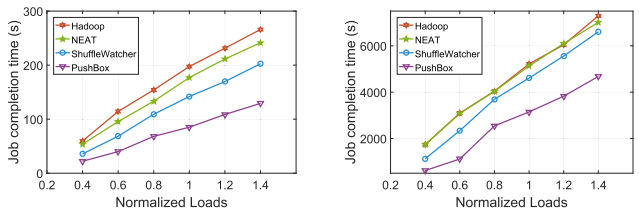
(a) Average completion time (b) 99% tail completion time

Fig. 11. Impact of cluster resource.



(a) Average completion time (b) 99% tail completion time

Fig. 12. Impact of slots per machine.



(a) Average completion time (b) 99% Tail completion time

Fig. 13. Impact of workload.

with Hadoop, ShuffleWatcher and NEAT, PushBox reduces the average job completion time by 22.7%-67.0%, 17.3%-43.0% and 13.2%-57.9%, respectively. Meanwhile, for the tail job completion time, PushBox is better than others by 24.9%-66.2%, 19.1%-57.9% and 23.77%-63.0%, respectively.

What if More Slots Per Machine. As shown in Fig. 12, we evaluate PushBox with different numbers of slots per machine. Here we only change the number of slots, hence network resource is unchanged. Results show that when slots are increasing, the job completion time of PushBox and ShuffleWatcher is gradually decreasing. It is intuitive because more tasks are allowed to execute concurrently. However, the job completion time of Hadoop and NEAT is increasing when the number of slots is larger than 12. This is counter-intuitive and we dig into the logs. Because of less bandwidth allocated to a specific task, larger coflow jobs have a longer job completion time. ShuffleWatcher has a worse average job completion time than others when there are 8 slots because slots are insufficient to transfer to help other jobs. Although PushBox also cannot transfer slots to help other jobs, a good network abstraction brings better performance. Compared with Hadoop, ShuffleWatcher and NEAT, PushBox reduces the average job completion time by 31.3%-57.5%, 22.9%-40.0% and 21.7%-53.1%, and tail job completion time by 4.5%-40.2%, 6.9%-33.0% and 4.9%-39.2%, respectively.

What if Increase Workload. As shown in Fig. 13, we evaluate PushBox with different workloads by scaling job size in the Facebook log and keeping the resources unchanged. Note that the workload is normalized by the original traffic load in figures, while network IO and computation are scaled with the same factor. For all evaluated algorithms, both the average and tail job completion time demonstrate a growing trend as traffic load increases. PushBox performs best all the time. Compared with Hadoop, ShuffleWatcher and NEAT, PushBox reduces the average job completion time by 51.3%-64.9%, 35.9%-41.5%, and 46.4%-58.7%, and tail job completion time by 33.8%-67.0%, 27.5%-56.3%, and 33.2%-66.8% respectively.

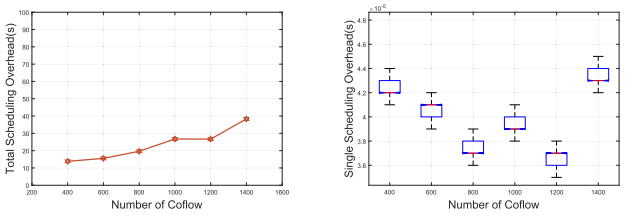
7.4 Scalability

Complexity Analysis. To demonstrate Pushbox's scalability, we analyze the complexity of the algorithm from a theoretical point of view. The key variables involved in the algorithm are first defined: the total number of job queues in a multi-job queue is p, the maximum number of tasks in all job queues is n, and the total number of corresponding machines is m. Then we perform a time complexity analysis for each of the critical functions in the algorithm.

- `selectTask` : the function has two branches, and the judgment of which branch to enter is based on whether tasks have the opportunities of locality. The time complexity of the judgment can be regarded as $\mathcal{O}(n)$. Pushbox will select a task according to the locality policy if the first branch is entered, and the time complexity can be considered $\mathcal{O}(n)$. Conversely, if it enters another branch, Pushbox will sort all waiting tasks in descending order of input size and select the first task. Due to the inclusion of sorting, the time complexity of this process is at least $\mathcal{O}(n\log n)$. Therefore the total time complexity of this function is $\mathcal{O}(n\log n)$.
- `canBePlaced` : this function determines if the selected task can be placed, containing only two comparisons so that the time complexity can be considered $\mathcal{O}(1)$.
- `notHinderTask` : this function determines whether the possible beneficiary tasks harm the network and contains only one comparison, so the time complexity is also $\mathcal{O}(1)$.

Specifically for the two scheduling models, the time and space complexity analyses are shown below.

- *Slot-change:* in this mode, PushBox traverses the p job queues in turn. In each iteration, the primary time overhead is in the calls to the three critical functions mentioned above. First, the list of tasks is traversed by calling `selectTask` to select a candidate task. If the candidate task is not legitimate, a `notHinderTask` test is called not to compromise bandwidth. If the



(a) Total Scheduling Overhead (b) Single Scheduling Overhead

Fig. 14. Scheduling Simulation Overhead.

task is a remote task with an input phase, PushBox will call `canBePlaced` to determine if it can be placed on this machine. Based on the above analysis, the time complexity of one traversal is $\mathcal{O}(n\log n)$ for a total of p rounds, so the total time complexity of the pattern is $\mathcal{O}(p \times n\log n)$. Since the algorithm does not use any additional storage space, the space complexity of the algorithm is $\mathcal{O}(1)$.

- *Task-arrival:* the primary time overhead of Task-arrival mode is concentrated on sorting all machines with free slots and the traversal of all machines. Pushbox needs to call the `selectTask` function for each machine traversal, with time complexity of $\mathcal{O}(n\log n)$. In contrast, the rest of the operation time overhead can be treated as a constant. Thus the m-round traversal time complexity is $\mathcal{O}(m \times n\log n)$ and the total pattern time complexity is $\mathcal{O}(m \times n\log n)$. The space complexity of this algorithm is $\mathcal{O}(\max(m,n))$ since the $taskMachinePair$ that records matching task-slot pairs occupies the space of $\max(m,n)$.

Scheduling Overhead. A theoretical analysis of the scalability of Pushbox is not enough. We also evaluate the algorithm scheduling overhead through simulation. Based on Sincronia [16], we generated Coflow workloads with the same distribution as the Facebook log. The scheduling overhead of Pushbox was measured by varying the number of Coflow of the input workloads while keeping the machine resources constant. The number of Coflow in the generated workloads scales from 400 to 1400. Fig. 14a shows the total overhead of Pushbox, while Fig. 14b shows a box plot of the single scheduling time for different workloads. Results show that Pushbox's total scheduling overhead is approximately linear concerning the workload, while Pushbox's single scheduling overhead remains stable at around 35 to 50 microseconds despite the increasing workload.

Summary. Through the above theoretical analysis and experimental evaluation, we can conclude that Pushbox has low complexity and stable single scheduling overhead, which shows good scalability.

8 Discussion

How to Get or Predict a Task's Input Size? A major concern of explicit network scheduling is the lack of accurate information on network footprint. For example, some researchers assume that network footprint is completely unknown a priori [15], [46]. This is not the case in many scenarios. Rich traffic demand information already exists in the log and meta-data files. It has been reported that for Hadoop and Spark, a flow's source, destination, and footprint can be nearly 100% accurately extracted when a computation stage finishes [25], [26]. Further, resource requirements of recurring business-critical jobs can be accurately predicted [11]. For example, the input/output data ratio is relatively stable for a specific application [44], [45]. There is much existing work that estimates job sizes. We rely on them to sort the priority order of jobs. Evaluations demonstrate that our approach is robust to estimation error in a large range.

Why Output is Not a Phase of Task Lifetime. For tasks other than the final stage, the output is modeled as input to the following stage's task input phase. For a task in the final stage, its output data is the final job result in memory. Even if the output needs persistence in distributed storage, the writing procedure is not in the critical path and can be detoured to idle network links [47].

Other Related Work. There are also many cluster scheduling works focusing on either fairness(e.g., [48]), or straggler mitigation(e.g., [49]). These researches are complementary to the objective of this paper. For example, a straggler task is always on the critical path [50]. PushBox gives priority in task selection for a straggler task.

Future Work. There are two directions. First, PushBox needs the extension to support heterogeneous scenarios. For example, heterogeneous machines exist in clusters due to the business purchase cycle. Also, different applications might need different containers with different configurations of CPU cores and memory. Dynamic slot partition support is required. Second, we are porting PushBox to Spark. We also hope to extend PushBox to support other data-parallel systems.

Acknowledgments

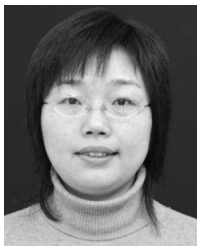
The authors would like to thank anonymous reviewers for their valuable comments.

References

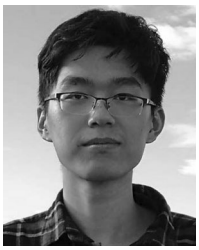
[1] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, 2013, pp. 69–84.
[2] A. Shinnar, D. Cunningham, V. Saraswat, and B. Herta, "M3R: Increased performance for in-memory hadoop jobs," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.
[3] J. Ekanayake *et al.*, "Twister: A runtime for iterative MapReduce," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 810–818.
[4] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 15–28.
[5] Apache hadoop. 2019. [Online]. Available: http://hadoop.apache.org
[6] V. K. Vavilapalli *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.
[7] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[8] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 261–276.
[9] G. Ananthanarayanan *et al.*, "Reining in the outliers in Map-Reduce clusters using Mantri," in *Proc. 9th USENIX Conf. Oper. Syst. Des. Implementation*, 2010, vol. 1, pp. 265–278.
[10] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "ShuffleWatcher: Shuffle-aware scheduling in multi-tenant Map-Reduce clusters," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2014, pp. 1–12.

[11] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 407–420.
[12] J. Tan *et al.*, "DynMR: Dynamic MapReduce with reduce task interleaving and maptask backfilling," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.
[13] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 431–442, 2014.
[14] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2014, pp. 443–454.
[15] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 393–406, 2015.
[16] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: Near-optimal network design for coflows," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 16–29.
[17] B. Tian *et al.*, "Scheduling dependent coflows to minimize the total weighted job completion time in datacenters," *Comput. Netw.*, vol. 158, pp. 193–205, 2019.
[18] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," in *Proc. ACM Conf. Special Int. Group Data Commun.*, vol. 42, no. 4, pp. 187–198, 2012.
[19] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proc. ACM Conf. Special Int. Group Data Commun.*, vol. 41, no. 4, pp. 242–253, 2011.
[20] K. Liu *et al.*, "Exploring token-oriented in-network prioritization in datacenter networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 5, pp. 1223–1238, May 2020.
[21] J. Zheng *et al.*, "Django: Bilateral coflow scheduling with predictive concurrent connections," *J. Parallel Distrib. Comput.*, vol. 152, pp. 45–56, 2021.
[22] A. Greenberg *et al.*, "VL2: A scalable and flexible data center network," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2009, vol. 39, pp. 51–62.
[23] A. Singh *et al.*, "Jupiter rising: A. decade of clos topologies and centralized control in Google's datacenter network," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 183–197.
[24] M. Zaharia, "Job scheduling with the fair and capacity schedulers," *Invited Talks in Hadoop Summit*, 2009.
[25] Y. Peng, K. Chen, G. Wang, W. Bai, Z. Ma, and L. Gu, "Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing," in *Proc. IEEE Conf. Comput. Commun.*, 2014 pp. 19–27.
[26] H. Wang *et al.*, "FLOWPROPHET: Generic and accurate traffic prediction for data-parallel cluster computing," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 349–358.
[27] R. Wilhelm *et al.*, "The worst-case execution-time problem-overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.
[28] H. Chang, M. Kodialam, R. R. Kompella, T. Lakshman, M. Lee, and S. Mukherjee, "Scheduling in MapReduce-like systems for fast completion time," in *Proc. IEEE Conf. Comput. Commun.*, 2011, pp. 3074–3082.
[29] F. Chen, M. Kodialam, and T. Lakshman, "Joint scheduling of processing and shuffle phases in MapReduce systems," in *Proc. IEEE Conf. Comput. Commun.*, 2012, pp. 1143–1151.
[30] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2011, vol. 41, pp. 98–109.
[31] F. Giroire, N. Huin, A. Tomassilli, and S. Pérennes, "When network matters: Data center scheduling with network tasks," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 2278–2286.
[32] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu, "Network scheduling aware task placement in datacenters," in *Proc. 12th Int. Conf. Emerg. Netw. EXperiments Technol.*, 2016, pp. 221–235.
[33] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in MapReduce applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 25–36.
[34] Q. Chen, J. Yao, and Z. Xiao, "LIBRA: Lightweight data skew mitigation in MapReduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2520–2533, Sep. 2015.
[35] B. Tian *et al.*, "Using the macroflow abstraction to minimize machine slot-time spent on networking in hadoop," in *Proc. 2nd Asia-Pacific Workshop Netw.*, 2018, pp. 36–42.
[36] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 5, pp. 506–521, May 1996.
[37] H. Huang and H. Shen, "Fairness-aware scheduling of dynamic cross-job coflows in shared datacenters based on meta learning," *Comput. Elect. Eng.*, vol. 100, 2022, Art. no. 107815.
[38] Puma benchmarks and dataset downloads. 2019. [Online]. Available: https://engineering.purdue.edu/puma/datasets.htm
[39] Coflow benchmark. 2019. [Online]. Available: https://github.com/coflow/coflow-benchmark
[40] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proc. 27th ACM Symp. Parallelism Algorithms Archit.*, 2015, pp. 294–303.
[41] X. S. Huang, X. S. Sun, and T. Ng, "Sunflow: Efficient optical circuit scheduling for coflows," in *Proc. 12th Int. Conf. Emerg. Netw. EXperiments Technol.*, 2016, pp. 297–311.
[42] Y. Li *et al.*, "Efficient online coflow routing and scheduling," in *Proc. 17th ACM Int. Symp. Mobile Ad Hoc Netw. Comput.*, 2016, pp. 161–170.
[43] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "CODA: Toward automatically identifying and scheduling coflows in the dark," in *Proc. Conf. ACM SIGCOMM Conf.*, 2016, pp. 160–173.
[44] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *Proc. IEEE 19th Annu. Int. Symp. Modelling Anal. Simul. Comput. Telecommun. Syst.*, 2011, pp. 390–399.
[45] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in Big Data systems: A cross-industry study of MapReduce workloads," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.
[46] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. 12th USENIX Conf. Netw. Syst. Des. Implementation*, 2015, pp. 455–468.
[47] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2013, vol. 43, pp. 231–242.
[48] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, vol. 11, pp. 323–336.
[49] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 2008, vol. 8, pp. 29–42.
[50] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, vol. 13, pp. 185–198.

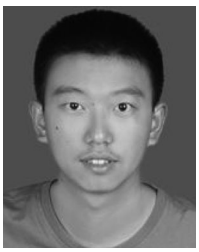
Chen Tian received the BS, MS, and PhD degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2008, respectively. He is a professor with State Key Laboratory for Novel Software Technology, Nanjing University, China. He was previously an associate professor with the School of Electronics Information and Communications, Huazhong University of Science and Technology, China. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University. His research interests include data center networks, network function virtualization, distributed systems, internet streaming, and urban computing.



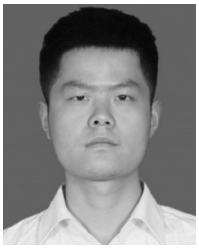
Yi Wang received the BS, MS, and PhD degrees from the Department of Electronics and Information Engineering, Huazhong University of Science and Technology, China, in 2000, 2003, and 2009 respectively. She is currently a lecturer with the School of Modern Posts, Nanjing University of Posts and Telecommunications, China. Her research interest is cloud computing.



Bingchuan Tian received the BE degree from the Department of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China, in 2016, and the PhD degree from the Department of Computer Science and Technology, Nanjing University, China, in 2021. His research interests include intent-based networking, congestion control, and network scheduling.



Yang Zhao received the master's degree from the Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2019. His research interests focus on datacenter network and wireless charging.



Yuhang Zhou received the BS degree from the Department of Computer Science and Technology, Nanjing University, China. He is currently working toward the MS degree with the Department of Computer Science and Technology, Nanjing University. His research interests include data center networks and distributed machine learning systems.

Chenxu Wang is currently working toward the BEng degree with the School of Artificial Intelligence, Nanjing University, China. His research interests include datacenter networks and data processing unit.

Haoran Guan is currently working toward the BS degree with the Major of Data Science and Major of Software Development, University of Sydney, Australia. His research interests include data analysis and machine learning systems.

Wanchun Dou received the PhD degree in mechanical and electronic engineering from the Nanjing University of Science and Technology, China, in 2001. He is currently a full professor of the State Key Laboratory for Novel Software Technology, Nanjing University. From April 2005 to June 2005 and from November 2008 to February 2009, he respectively visited the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, as a visiting scholar. Up to now, he has chaired three National Natural Science Foundation of China projects and published more than 60 research papers in international journals and international conferences. His research interests include workflow, cloud computing, and service computing.

Guihai Chen received the BS degree in computer software from Nanjing University, in 1984, the ME degree in computer applications from Southeast University, in 1987, and the PhD degree in computer science from the University of Hong Kong, in 1997. He is a distinguished professor of Nanjing University. He had been invited as a visiting professor by Kyushu Institute of Technology in Japan, University of Queensland in Australia and Wayne State University. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture and data engineering. He has published more than 350 peer-reviewed papers, and more than 200 of them are in well-archived international journals such as *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE/ACM Transactions on Networking* and *ACM Transactions on Sensor Networks*, and also in well-known conference proceedings such as HPCA, MOBIHOC, INFOCOM, ICNP, ICDCS, CoNext, and AAAI. He has won 9 paper awards including ICNP 2015 Best Paper Award and DASFAA 2017 Best Paper Award.